THE FORMAL SPECIFICATION OF THE TEES CONFIDENTIALITY MODEL

ANTHONY HOWITT

A thesis submitted in partial fulfilment of the
requirements of the University of Teesside
for the degree of Doctor of Philosophy

September 2008

Abstract

This thesis reports an investigation into authorisation models, as used in identity and access management. It proposes new versions of an authorisation model, the Tees Confidentiality Model (TCM), and presents formal specifications in B, and verifications and implementations of the key concepts using Spec Explorer, Spec# and LinQ.

After introducing the concepts of authorisation and formal models, a formal methods specification in B of Role Based Access Control (RBAC) is presented. The concepts in RBAC have heavily influenced authorisation over the last two decades, and most of the research has been with their continued development. A complete re-working of the ANSI RBAC Standard is developed in B, which highlights errors and deficiencies in the ANSI Standard and confirms that B is a suitable method for the specification of access control.

A formal specification of the TCM in B is then developed. The TCM supports authorisation by multiple concepts, with no extra emphasis given to Role (as in RBAC). The conceptual framework of Reference Model and Functional Specification used in the ANSI RBAC Standard is used to structure the TCM formal model. Several improvements to the original TCM are present in the formal specification, notably a simplified treatment of collections. This new variation is called TCM2, to distinguish it from the original model.

Following this, a further B formal specification of a TCM reduced to its essential fundamental components (referred to as TCM3) was produced. Spec Explorer was used to animate this specification, and as a step towards implementation

An implementation of TCM3 using LinQ and SQL is then presented, and the original motivating healthcare scenario is used as an illustration.

Finally, classes to implement the versions of the TCM models developed in the thesis are designed and implemented. These classes enable the TCM to be implemented in any authorisation scenario.

Throughout the thesis, model explorations, animations, and implementations are illustrated by SQL, C# and Spec# code fragments. These illustrate the correspondence of the B specification to the model design and implementation, and the effectiveness of using formal specification to provide robust code.

Acknowledgments

I would like to express my sincere thanks to my supervisors. To Dr. Jim Longstaff for his encouragement and unfailing good humour throughout the course of this project. And to Steve Dunne for rekindling an enthusiasm for mathematics and its application that had long lain dormant.

I would also like to thank the School of Computing for the provision of finance, and for the support and encouragement of many within it, particularly Zafar Khan, Erika Downs, Annette Marshall, and Carol Mooney.

In addition, to give thanks in general for the access the University of Teesside has given me to Higher Education in these later years of my life.

Following Dr. Longstaff's example in his thesis, I include a favourite piece of scripture:

"Now to him who is able to do immeasurably more than all we ask or imagine, according to his power that is at work within us, to him be glory in the church and in Christ Jesus throughout all generations, for ever and ever! Amen."

Ephesians 3:20,21

Table of Contents

Table of Figures

Table of Screenshots

Tables

1 Introduction

This thesis is concerned with research which has led to improved versions of the Tees Confidentiality Model (TCM), which is an access control model developed by the University of Teesside to provide authorisation in complex scenarios. The TCM had previously been developed using an iterative prototyping method, with each stage involving modelling, implementation and feedback. It was felt that a more rigorous treatment was needed to fully explore the issues, and that such a treatment would very likely lead to improvements to the model.

The initial main idea was to apply formal methods to TCM to ensure that the TCM was rigorously defined, better understood, and without any inconsistencies.

1.1 Aims and Objectives

At the outset of the research programme, the overall aims and objectives for the research programme were as follows.

- To conduct a literature review and investigation into the fundamental concepts of authorisation, and authorisation models.
- To determine whether formal methods, and in particular B, could provide a formal specification of the TCM
- To compare and contrast different methods of formal specification
- To create a formal model of the TCM as a basis for software implementation
- To develop software tools to enable the implementation of the TCM in complex authorisation scenarios

The research would be informed by previous TCM research and development projects, and the resulting academic publications. In particular, scenarios suggested by the health service would be used to illustrate the research results.

The formal methods which would be used are the B-Method, Spec Explorer, and Spec#.

Formal methods provide the kind of evidence that is needed in high-risk industries such as aviation. They demonstrate responsible engineering and give solid reasons for trust in the product. (Hall, 2006). The development of software for access control is an area where high integrity is required e.g. in access to online health records, which was a starting point for the development of the TCM.

During the initial literature review stages of the project, it was realised that the mathematical model forming the basis of Role-Based Access Control (INCITS 2004) could provide the starting point for the TCM, and for other models of authorisation.

B was successfully applied to the RBAC model, something that had never been attempted before. It was so successful that a number of errors were highlighted in the ANSI RBAC standard. It was also realised that B was a powerful tool for modelling not just RBAC and the TCM but the fundamental basis of access control itself.

The aims and objectives stated above were revised accordingly:

- To conduct a literature review and investigation into the fundamental concepts of authorisation, and authorisation models.
- To investigate a basic model of authorisation using the ANSI RBAC standard as a starting point.
- To produce a formal specification of the well-established ANSI RBAC model, as a precursor to investigating the TCM and possibly other authorisation models
- To demonstrate that this specification was an improvement over the specification in the standard.
- To apply the B techniques learnt in the RBAC B specification to the rules and principles of the much more sophisticated TCM to produce a formal model

The power of B was such that further objectives were added during the course of the TCM formal specification:

- To simplify the mathematical basis of the TCM whilst keeping the power of the original model.
- To formally specify generalisations of the TCM by removing or simplifying constraints in the model.
- To specify a generalisation of RBAC which consists of the fundamental principles of the TCM
- To develop tools to enable general authorisation models, including the TCM, to be applied to any scenario.

1.2 Introduction to Access Control

Access control has existed as long as human beings had assets they wanted to protect. Locks, gates, and guards are all forms of access control. Access management systems are usually perceived as consisting of three parts: authentication, for establishing the identity of the user; authorisation, for determining the resources that the user is permitted to use; and administration. This thesis is mainly concerned with the second of these: authorisation.

Access control is needed in virtually all systems, and imposes architectural and administrative challenges at all levels of software development and implementation. From a business perspective, access control has the potential to optimise sharing and distribution of resources. However, if poorly applied it can frustrate users by denying access to necessary resources, or by requiring elaborate administrative procedures with corresponding costs. If applied incorrectly it can cause the unauthorised disclosure or corruption of valuable data.

In a 1987 paper (Wilson, et al., 1987) Wilson and Clark argued that while security was important to commercial users, their primary concern was data integrity i.e. that data could only be modified in certain ways by particular users. It was to address this need that Role-based access control (RBAC) was developed.

1.3 History of Access Control

Computer access control and security became increasingly important in the 1970s with the increase of large resource sharing systems in defence and large commercial organisations. The use of ATMs and other automated systems also increased the need for strong security.

In the early 1980s the US Department of Defence defined in detail two important methods of access control for military systems; discretionary access control (DAC) and mandatory access control (MAC) (Department of Defence, 1985).

Discretionary Access Control is such that the creator or owner of an object can assign access rights to that object, and anyone with discretionary access to an object can pass those rights to another user. However, with DAC, there is no way to be sure that some unauthorised user will not eventually receive inappropriate rights through some chain of delegation. To provide a secure system MAC was required. MAC"s most important feature was to deny users full control over the resources they create. The system security policy (as set by the administrator) entirely determines the access rights granted, and a user

may not grant less restrictive access to their resources than the administrator specifies. For MAC, the access control decision is contingent on verifying the compatibility of the security properties of the data and the clearance properties of the individual (or the process proxying for the individual).

The following list includes some of the most important security features, currently required by the U.S. Department of Defence (Department of Defence, 1985):

- It must be possible to control access to a resource by granting or denying access to individual users or named groups of users.
- Memory must be protected so that its contents cannot be read after a process frees it. Similarly, a secure file system, such as NTFS, must protect deleted files from being read.
- Users must identify themselves in a unique manner, such as by password, when they log on. All auditable actions must identify the user performing the action.
- System administrators must be able to audit security-related events. However, access to the security-related events audit data must be limited to authorised administrators.
- The system must be protected from external interference or tampering, such as modification of the running system or of system files stored on disk.

1.3.1 Authorisation Models

An authorisation model is concerned with ways in which users can access resources in a computer system. Informally "who is allowed to do what to what?" Users can be human end-users or other computer systems. The operations performed on objects range from simple querying of data, to sophisticated application facilities.

The simplest form of authorisation is an access control list (ACL). The ACL contains the names of objects and users, together with the specific operations that the user is authorised to perform on the object. If a user wants to perform an operation on an object, the system searches for an entry on the ACL. If the appropriate entry exists then the operation is allowed.

User	**Operation**	**Object**
Bob	Read, Write	File A

Bob	Read, Execute	File B
Jim	Read	File C
Jim	Modify	File D

Any authorisation model (no matter how sophisticated) can be represented and implemented as an ACL. The problem when dealing directly with an ACL is the allocation and maintenance of what can become a very large number of entries. The main function of an authorisation model then becomes to maximise administrative efficiency whilst allowing sufficient granularity in the granting of permissions. In this area, role-based access control (RBAC) has been predominant for a number of years.

Administrative efficiency comes from assigning permissions to groups or collections of users. In RBAC, permissions are assigned to roles, and then collections of users are assigned to each role. In this way, a collection of users can be assigned to a collection of permissions. Inheritance in RBAC allows an even larger collection of users to be assigned to a collection of permissions, because the collections of users assigned to any descendant roles are also included.

However, in many cases the granularity has proved insufficient and occasioned the development of extensions to RBAC e.g. „parameterised RBAC" (Ge, et al., 2004), which uses role plus some other factor, such as location. The TCM takes the idea of using collections for administrative efficiency a step further. Essentially, an authorisation model uses some form of collections to connect users, operations, and objects to simplify administration. In the TCM, this is acknowledged „up front" and „role" is not given a special place. The TCM allows collections of operations and objects as well as users, and uses classifiers and classifier values to facilitate that.

1.3.2 Context

An authorisation model, through its implementation within an identity and access management system, provides facilities to enable users, whether they are human end-users or other computer systems, to use resources in specified ways.

The Tees Confidentiality Model (Longstaff, et al., 2003) (Longstaff, et al., 2006) is a powerful model for authorisation, and is unique in that it includes comprehensive override capabilities. It can support traditional discretionary access control (DAC), mandatory access control (MAC) and RBAC by representing each with a TCM permission type.

Other concepts such as credentials and trust can be supported within the same framework. The TCM lends itself to implementation by database systems, and I develop elements of its implementation by Microsoft Transact SQL, and other means.

1.3.3 Identity Management Infrastructures

For large applications, the design of authorisation models must cater for federated identity management.

Federated Network Identity is the concept of utilising distributed identity stores, with no central management of identity. In the federated model, network identity and user information is distributed across a number of locations. Each participant in the federation agrees common standards for authentication and authorising each other‟s users. This is the model used for the Project Liberty initiative (Project Liberty, 2006), which itself uses the SAML standard (Organization for the Advancement of Structured Information Standards, 2006). Project Liberty has produced extensive models of trusted environments. The SAML standard specifies how one entity passes authentication and authorisation information to other using XML internet standards.

Another language is XACML, which facilitates the defining of fine-grained authorisations (see (Mazzoleni, et al., 2006) for a discussion and application of XACML). These mechanisms permit the signing and encryption of parts of XML documents. The authorisation model presented in this paper could be used within federated infrastructures to support powerful authorisation functionality, defined in readily understandable and essentially simple terms.

Another characteristic of modern computing is the increase in mobile device usage. More users will connect to the organisation‟s resources from outside the firewall e.g. working at home, using a laptop, using a PDA. Many users will not be full members of the organisation – they might be employed in joint ventures, or might be the staff of partners, or contracting staff, or the staff of customers or suppliers. Authorisation policies must cater for these situations.

Finally, it must be stated that the implementation of authorisation will often take place in a service oriented computing infrastructure, using web services. Therefore, the applications designer should be aware of these concepts, their standards and implementation techniques.

1.3.4 Role-Based Access Control

The most widely used model of authorisation is role-based access control (RBAC), and one of the most comprehensive works on the RBAC model is by Ferraiolo et al. (Ferraiola, et al., 2001). RBAC has been used in computing practice such as operating systems, databases, web access management, and more recently in Service Oriented Computing applications. Extensive research literature on RBAC can be found from the ACM SACMAT and TISSEC web portals and elsewhere; these sources also provide many examples of the use of RBAC in healthcare and commercial applications. A summary of the commercial and industrial application of RBAC can be found in an EEMA whitepaper (The Independent European Association for eBusiness, 2003). An ANSI INCITS standard for RBAC exists (INCITS, 2004).

The basic principle of the established RBAC model of authorisation is that users acquire permissions through being assigned to roles. Here "permission" means the granting of authority to perform an operation on a protected object (i.e. a resource), e.g. the granting of read access for part of a patient‟s medical record. This contrasts with identity-based Access Control (IBAC) in which permissions are assigned directly to users. The IBAC approach suffers from problems resulting from large numbers of permission assignments; the main purpose of RBAC is to reduce the number of permission assignments, and to facilitate their management (Bacon, et al., 2001).

RBAC generally provides the following benefits:

- Increased scalability for numbers of users and applications, especially for web-based users.
- Improved productivity and efficiency through speed of response to organisational change, timely availability of resources to authorised users, and delegated authority.
- Separation of duties (a user cannot inappropriately activate two or more roles at the same time).
- Principle of least privilege – users only have the authorisations they actually need.
- „Extended enterprise‟ benefits: development of trust relationships between organizations based on role model mappings. In addition, RBAC can be one of the bases of shared methods for authentication and authorisation in federated infrastructures.

1.3.5 The Basis of the TCM

This section gives a description of the principle, which is novel to the original TCM, and to the improved versions developed in this thesis.

In basic RBAC, authorisation is determined by the single concept of role. Each role is associated with one or more permissions, which enable an activity to take place. Therefore, a role, e.g. „GP" can be associated with a permission that allows reading and updating a patient"s medical record. In the TCM, authorisation is determined by multiple concepts, of which role may be one. No special significance is attached to role, indeed it could be absent if the authorisation policy required it. In addition, the equivalent of RBAC can be expressed in the TCM.

The mechanism in the TCM involves a new type of permission, which is called a Confidentiality Permission CP, to distinguish it from permissions as used in RBAC. Each concept to be used in authorisation is specified as a Classifier, and although at the start of this research a CP had a much more complex structure (see 1.3.6 below), a CP essentially consists of a set of classifiers and associated values. For example, a CP might contain the following classifier/ classifier value pairs:

{<Identity, Fred>, <Role, GP>, <OperationType, Read>, < HealthRecordID, EHR1>}

This CP would allow Fred, if he was acting in a GP role, read access to health record EHR1. (Much greater subtlety of permission is possible than is shown in this simple example: this is developed later in this thesis.)

1.3.6 Overview of the Different TCM Models

In this section I briefly describe how the use of formal methods led to the improvements of the TCM and give a description of the different versions of the TCM. It is an account of the evolution of the access control research, fully described in the main sections of this thesis.

Essential to the TCM as defined in a variety of papers (Longstaff, et al., 2006) (Longstaff, et al., 2003) at the start of this thesis (I shall call this version TCM1) is a range of permission types, called Confidentiality Permission Types (CPTs), which are processed in a defined order. With each CPT there are associated Confidentiality Permissions that may have negative values (i.e. they may deny access), and may be overridden by authorised identities in carefully specified ways.

A Confidentiality Permission Type consists of a set of classifiers that together are considered useful for authorisation e.g.

{Identity, Role, OperationType, HealthRecordID}

N.B. although this notation is different from that used in the original papers it is essentially the same, except that the original notation enforced the TCM1 requirement that there be at least one of each set of user, operation, and object classifiers in a CPT. This requirement was dropped for TCM3.

In TCM1 a single concept of Collection is used for structuring classifier values, including roles. Confidentiality Permissions were defined using these collections and included the specification of inheritance within their definition. This inheritance could be upward or downward through the collections.

The application of formal specification to TCM1 brought about a number of rationalisations to the model. It was realised that:

- There was no need for upward and downward inheritance because every mixture of upward and downward inheritance applied to different collections within a CP, could be rewritten as all downwards inheritance.
- The concept of collections itself was duplicating and overlapping the concept of classifiers i.e. an identity (or operation, or object) could be „classified" as belonging to a collection. Thus, inheritance could be applied directly to the classifier values (as in RBAC).
- There was no need for different inheritance relationships for different classifiers (e.g. role and location) but all inheritance for all classifiers (including operation and object classifiers could be modelled by a single inheritance relationship.

TCM2 is TCM1 with the rationalisations as detailed above applied. The inner workings of TCM2 are thus much simpler than those of TCM1. Although CPTs and CPs could still be created through a user interface as in TCM1 (i.e. using collections, and upward and downward inheritance), these would be translated to an internal model based on TCM2.

Further, it was realised that the TCM (in both the TCM1 and TCM2 specification) was in fact a generalisation of RBAC. It was natural to ask which of the features of the TCM were part of a first step generalisation, and which were additional (perhaps unnecessary) extra add-ons. It also became clear that the TCM had something to say about the

fundamental workings of an authorisation system in general. This led to the development of TCM3.

TCM3 is described in parts of this thesis as a simplification of TCM2. It would be more correct to describe it as both a generalisation and a simplification of TCM2 which I believe has all the power of TCM2 without some of the extra complications. The following features of TCM2 are not included in TCM3:

- TCM2 ties every CP to a fixed set of CPTs. Thus, the CPTs act as a constraint on what can be considered a valid CP. In addition, there must be at least one of each set of user, operation, and object classifiers. For TCM3 any set of <classifier, value> pairs can be a CP e.g. just {<Role, Administrator>}. This is discussed more fully later, but this CP would allow any user in the role of Administrator to perform any operation on any object. Interesting questions such as what would a CP equal to the empty set do? It can be shown that this allows any user to perform any operation on any object. If it were a prohibition it would prevent all operations.
- TCM2 has the concept of CPT ordering which effectively makes some CPs more important than others. This is thought valuable in order to implement the override features of the TCM. In TCM3 we say that every CP has equal value, and explore the implementation of override through a system classifier.

TCM3 is fully discussed in the relevant section (4). It is a generalisation of RBAC and of TCM2 which I believe contains valuable mechanisms applicable to all authorisation systems.

1.4 Introduction to Formal Methods

Formal methods are used to provide software without bugs, rather than write software with lots of bugs and then spending time removing bugs from it. Programs written using formal methods can be considerably smaller than those without. The application of mathematics enforces compactness and non-duplication. It is intended to demonstrate this with the TCM.

The application of formal methods does delay coding. In addition, much of the recent development approach has concentrated on coding very quickly, and in general, this has produced good results, even though many bugs have to be sorted out later through service packs and patches.

However, formal methods are heavily used in the development of military systems, and in safety-critical software. B was used in the development of signalling software for the French railway system.

Microsoft, one of the main proponents of rapid application development has lately been taking formal methods more seriously, as can be shown by their employment of Sir Tony Hoare. The company developed a tool called Abstract State Machines Language. The current version of this is applied through Spec Explorer and will be applied to the TCM, as an alternative to the B-Method. Spec# is current Microsoft research, which applies formal methods as an extension of C#. This will be applied as part of the development of TCM classes in C#.

1.4.1 Overview of the B-Method

The B-Method (Abrial, 1996) is a formal approach to the specification and development of computing that draws together advances in formal methods spanning the last thirty years. It is based on a wide-spectrum pseudo-programming notation, the Abstract Machine Notation (AMN), which provides a common framework for the construction of specifications, refinements, and implementations. More importantly, it permits the formal verification of such systems. (Schneider, 2001)

The B-Method is recommended for safety-critical software because it allows proof obligations to be discharged, thus validating the software for the entire domain. (For a definition of "proof obligations", see 2.3.6.) This is contrasted with normal software testing, which can show up the presence of bugs, but cannot conclusively prove their absence. A B specification can be faithfully implemented in code, by means of refinement and other techniques.

The B notation places an emphasis on simplicity: it deliberately rules out complex programming constructs, forcing the designer to use clear and well-understood program statements. An example of the mathematics found in the ANSI INCITS standard for RBAC (INCITS, 2004), and said to be a form of Z, is given in Figure 1 below. Specification in B compares favourably with this, as will be demonstrated later in this thesis.

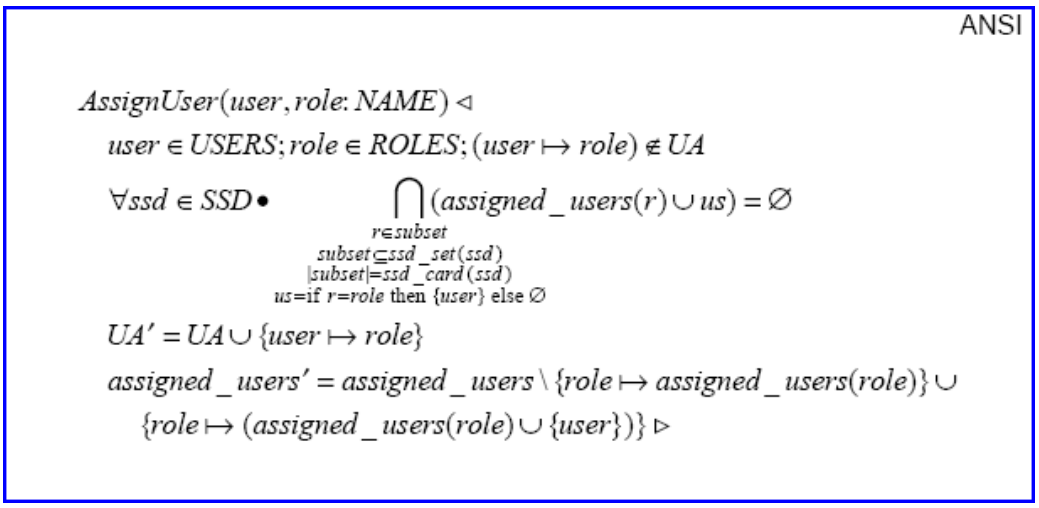


Figure 1: Maths in the RBAC Standard

In B systems are modelled as a collection of interdependent abstract machines, for which an object-based approach is employed at all stages of development.

An abstract machine is described using the Abstract Machine Notation (AMN). A uniform notation is used at all levels of description, from specification, through design, to implementation. Large machines can be constructed from other machines using the includes, uses and sees constructs.

A B-development contains invariant assertions that provide consistency conditions within and between components, such as machines. These invariants hold the development together and give rise to proof obligations that can be used to guarantee its correctness.

The B-Method further prescribes how to structure large designs and large developments, and promotes the re-use of specification models and software modules. Individual pieces of software systems are easy to understand, enabling verification of their combination and relationships with each other.

The B-Toolkit (B-Core) is a configuration tool that manages developments under the B-Method, generating proof obligations and supplying supporting tools for the discharge of those proof obligations (see 2.3.6). There is also support for the generation of documentation, and for the browsing of developments.

1.4.2 Spec Explorer

Spec Explorer (Campbell, et al., 2005) is a software development tool for advanced model-based specification and conformance testing. Spec Explorer can help software

development teams detect errors in the design, specification, and implementation of their systems. The tool is intended to be used by software testers, designers, and implementers. The core ideas behind Spec Explorer are:

- To encode a system's intended behaviour (its specification) in machine-executable form (as a "model program"). The model program typically does much less than the implementation; it does just enough to capture the relevant states of the system and show the constraints that a correct implementation must follow. The goal is to specify from a chosen viewpoint what the system must do, what it may do and what it must not do.
- To explore the possible runs of the specification-program as a way to systematically generate test suites.
- To compare the behaviour of the model program to the system's implementation in each of the scenarios discovered by algorithmic exploration.

Spec Explorer consists of the following components:

- The software modelling languages Spec# and AsmL.
- An explicit-state model checker, which allows the user to search the space of all possible sequences of method invocations that 1) do not violate the pre- and post-conditions and invariants of the system's contracts and 2) are relevant to a user-specified set of test properties.
- A traversal engine, which unwinds the resulting finite state machine to produce behavioural tests that cover all explored transitions.
- A binding mechanism allows users to associate actions of the model with methods of an implementation written .NET language.
- A conformance checker that executes the generated behavioural tests.

1.4.3 Spec#

The Spec# programming system (Barnett, et al., 2004) is a new attempt at a more cost effective way to develop and maintain high-quality software. Spec# is pronounced "Spec sharp" and can be written (and searched for) as the "specsharp" or "Spec# programming system". The Spec# system consists of:

- The Spec# programming language. Spec# is an extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions. It provides method contracts in the form of pre- and post-conditions as well as object invariants.
- The Spec# compiler. Integrated into the Microsoft Visual Studio development environment for the .NET platform, the compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools.
- The Spec# static program verifier. This component (codenamed Boogie) generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.
- A unique feature of the Spec# programming system is its guarantee of maintaining invariants (Barnett, et al., 2004) in object-oriented programs in the presence of callbacks, threads, and inter-object relationships.

1.4.4 .NET Language-Integrated Query (LinQ)

.NET Language-Integrated Query defines a set of general-purpose standard query operators that allow traversal, filter, and projection operations to be expressed in a direct yet declarative way in any .NET-based programming language. The standard query operators allow queries to be applied to any IEnumerable<T>-based information source i.e. an information source with a mechanism to iterate over the elements of a sequence, generally with the ultimate goal of applying to such sequences the **foreach** programming pattern. LINQ allows third parties to augment the set of standard query operators with new domain-specific operators that are appropriate for the target domain or technology.

More importantly, third parties are also free to replace the standard query operators with their own implementations that provide additional services such as remote evaluation, query translation, optimisation, and so on. By adhering to the conventions of the LINQ pattern, such implementations enjoy the same language integration and tool support as the standard query operators.

The extensibility of the query architecture is used in the LINQ project itself to provide implementations that work over both XML and SQL data. The query operators over XML

(LINQ to XML) use an efficient, easy-to-use, in-memory XML facility to provide XPath/XQuery functionality in the host programming language.

The query operators over relational data (LINQ to SQL) build on the integration of SQL-based schema definitions into the common language runtime (CLR) type system. This integration provides strong typing over relational data while retaining the expressive power of the relational model and the performance of query evaluation directly in the underlying store. (For further reading in this area, see (Rattz, 2007))

1.5 The Development of the TCM

Prior to the commencement of the research project (in October 2004) which has resulted in this thesis, the TCM had been developed as follows:

In 1998, a project started to model Electronic Patient Records (EPR). The participants were Dr Jim Longstaff (coordinator), Graham Capper, Professor Mike Lockyer from the University of Teesside; Mr Michael Thick (then a transplant surgeon and head of the Liver Transplant Unit at the Freeman Hospital, Newcastle-upon-Tyne); and Dr David Jones (from the NHS Information Authority).

Authorisation was recognised as a key issue, and some key concepts of the TCM appeared in the early UML models. The basic illustrative scenario (3.1.1) was devised by Mr. Thick, and the concept of overriding was devised by Dr. Longstaff. The first implementations were developed by Dr. Longstaff in 1999 (Ontos OO database, and also SQL Server). Due to internal promotion within the NHS by Mr. Thick and Dr. Jones, and several workshop and conference presentations, mostly by Dr. Longstaff, the TCM became recognised as being able to handle the most demanding authorisation scenarios in healthcare.

In 2000, the Tees Health Authority successfully bid for ERDIP funding, with patient confidentiality, based on the TCM, as one of the three main projects. Dr. Longstaff made a presentation of the SQL TCM demonstrator to the visiting ERDIP evaluation panel during the bidding process. Over the next three years, considerable recognition was afforded to the TCM, with several other ERDIP projects implementing their own versions of it. Such was the influence of the TCM that one bid for developing the Care Records System listed a TCM implementation as part of the proposed work programme. A further major project at Teesside University was directly funded by the NHSIA. This was the Health Records Infrastructure Programme project, for which Dr. Longstaff and Professor Lockyer

implemented a TCM demonstration on NHS computers at the Tees Health Authority, and this was accessed by other HRI projects in England. Subsequently, many presentations were given to the NHS National Programme for IT - NPfIT, with the general objective of showing how complex scenarios could be modelled and implemented.

Several high profile publications were also achieved, with very favourable feedback being obtained from leading academics and industry researchers.

Despite its success up until the start of this research project, there were a number of weaknesses that the developers recognised. The TCM was overly complex, with too many options available to model and implement authorisation scenarios. This was particularly true in the notion of collections. It was hoped that this project would identify and simplify the basic modelling and processing concepts. This has actually been achieved, and the results of this research project, reported in section 3 onwards, have very much influenced the most recent practical application of the TCM (by Dr. Longstaff), which I now briefly summarise.

On the basis of TCM publications, Dr Longstaff was approached by Private Access LLC, a start-up company in the USA (www.PrivateAccess.info). This company is developing a range of tools and services which have a Privacy Preferences Database (so called) as its core technology. This will be a central repository for patient‟s (Consumers) privacy requirements for their health data, and will be used by Health Record Holders to verify that disclosures of health data to Record Seekers are in accordance with the patients‟ requirements. (In the USA, there are many independent healthcare providers, who regularly have to process requests for health data from other providers, insurance companies, and other Record Seekers.) The Privacy Preferences Database will have other uses, such as facilitating participation in clinical trials. The TCM is being investigated as the model for the Privacy Preferences Database.

It soon became apparent that the TCM as previously published would be unable to handle the complexity of this new application. In the TCM prototype that has been developed for this application by Dr Longstaff, two key results of my research project as reported in this thesis were used. These were firstly the simplification of Collections, and secondly the use of partially completed permissions, with much less emphasis on type. Therefore, the present Privacy Preference Database prototype directly uses concepts derived from

theoretical considerations carried out during this project, and described in sections 3 and following below.

1.6 Authorisation Models for Health Informatics

1.6.1 Health Informatics Systems

There are many computerised health and social care records systems, with some major new developments. The NHS Care Records System (NHS Portal) (NHS Care Records), under development since 2004, is a major part of the largest non-military IT project in the world. In the USA, several high profile „Personal Health Records" systems (AHIMA) are already in use, and are being further developed: these include Microsoft HealthVault (HealthVault), and Indivo Health (Indivo) (Simons, et al.). A definition of a personal health record (PHR) taken from the AHIMA site is given in Appendix B.

One major function of PHRs is that they can import data from other providers, e.g. laboratory test data. In addition, PHRs can import data from other PHR systems. For example, HealthVault can import data from Indivo.

A less-ambitious PHR-equivalent in England is HealthSpace (Healthspace), which is a secure website where patients can store their personal health information; however HealthSpace will eventually provide a patients" interface to their Summary Care Record (held in the NHS Care Records System).

A key issue for all health informatics systems is the national infrastructure that supports them. The NHS in England and Wales has a central registry system for patients, and a unique patient identifier (the NHS Number). In addition, there is a central registry for doctors and other health care professionals. No equivalent registries exist in the USA. Dr. Longstaff recently visited the Indivo Health team at Harvard Medical School, and was told that one of the biggest obstacles to the development of data sharing and request facilities in the Indivo system is validating the requestor – it is difficult to know whether the requestor is a legitimate doctor because of the lack of a central registry.

Authorised health care professionals are only able to access patient records through the NHS Care Records Service once they have satisfied robust access control mechanisms. These include:

- A strong authentication process using smartcards that have chip and pass code technology, without which the system is inaccessible.

- They must have a Legitimate Relationship with any given patient (for example, be their registered GP or have been referred by an authenticated NHS clinician) before that patient record can be viewed.
- Role Based Access Control limits what functions can be used within a given application. For example the ability to view sealed data with patient permission.

Many hospital and GP systems have variants of Role Based Access Control as the authorisation component of their security systems. They are often supplied with a small number of pre-defined roles, e.g. GP, Senior GP, Receptionist, Administrator. Extra roles can be defined and implemented as required.

1.6.2 The NHS CRS Authorisation Model

In 2004, the first writings on an authorisation model for the CRS appeared on NHS Connecting for Health websites. It was described as the „Sealed Envelope" model, and it is the stated intention of the NHS to provide this during 2008 as part of the Care Records Service. The basic concepts of „Sealing", and „Sealing and Locking" data has not changed since 2004, though their presentation on NHS websites often has. There is a direct equivalent of the TCM concept of override, as researched as part of the NHS-funded ERDIP and HRI Programmes (see section 1.5 above), in the Sealed Envelope model.

The assumptions for the Sealed Envelope model are that

- Clinicians work in teams/work groups
- Confidential information needs to be shared within the team and protected from wider access
- Sealed Information is not seen by administrative roles unless local customisation redefines for particular roles and with appropriate training (RBAC)

There are two levels of sensitivity:

- A sensitive level – "sealed"
- A sensitive invisible level that is locked down and not available outside the clinical team/workgroup – "sealed and locked"

Clinicians in the workgroup in which the sealed data was created can see this data, although it is marked as sensitive. However, clinicians in another workgroup in the same organisation, or clinicians in a separate organisation, would have to „break the seal",

meaning invoking an override, in order to access the data. The existence of sealed data would have been communicated to them by means of a message or flag; it is up to their professional judgement whether they break the seal or not.

For sealed and locked data, clinicians in the workgroup in which the data was created can access the data, which is again marked as sensitive. However, members of other workgroups, in the same or other organisations, cannot access the data, and are not even aware of its existence – no message or flag is displayed to them when they access the patient"s record.

It is worth noting that an almost identical concept, called „Breaking the Glass", is being implemented as part of the Canadian Infoway Programme (Infoway, 2005). The TCM Model is referenced in this reference (Infoway, 2005).

1.6.3 The Indivo Health Sharing Model

Indivo health (Indivo) provides a simple sharing model, inviting users to choose one of five pre-installed authorisation policies. Sharing can only take place with a person who already has an Indivo account. The sharing policies are as follows.

- Primary Care Provider (allows reading of record, and adding of comments and updates)
- Family Member (allows reading of record)
- School (allows a university to read immunisation records)
- Research Administrator (allows this user to access data for research projects)
- Friend (allows reading of contact information, and sending of secure messages)

No fine-grained authorisation, as proposed for the Sealed Envelope Model, or supported by the TCM, is available in the Indivo PHR.

1.7 Overview of Thesis

Chapter 2 is a complete specification of the RBAC ANSI standard using B. The published standard uses a form of Z. It contains several errors. The B specification is a distinct improvement, and provides a better basis for programming. This chapter really needs to be studied together with the original documents (Ferraiola, et al., 2001) (INCITS, 2004) as the main area of innovation here is in the application of the B-Method. This Chapter also

includes some Spec# to demonstrate how the pre- and post-conditions from the B specification can be applied in a programming environment.

Chapter 3 is a complete specification of the TCM following the same format as the RBAC standard. Several improvements to the TCM as originally published are made, and the resulting (improved) model is called TCM2. TCM2 represents the state of the TCM at the present time i.e. as it has been developed through a series of papers, and as it has been applied by Dr. Longstaff in a number of real-life scenarios. Spec Explorer is introduced as another method of formal specification. The different methods are contrasted and compared. Some SQL is shown to demonstrate the link between the formal specification and the code. A demonstration application is produced which demonstrates some of the key features of TCM2.

Chapter 4 introduces a generalised and simplified version of TCM2, which for convenience I call TCM3. This retains the key features of the TCM whilst making it more accessible to administration. A justification for this is provided. The main operations (methods, functions) are described in both B and Spec#. TCM3 could be described as the bare bones of the TCM and it is demonstrated that even in this bare-bone state it represents a significant improvement to RBAC, of which it is a generalisation. It is shown that this generalisation gives increased flexibility and power over RBAC. Features of TCM2 which are not included in this bare-bone version such as Confidentiality Permission Type ordering can be added back if thought desirable in a particular situation. Some of the features of TCM3 are being used by Dr. Longstaff in his current work with Private Access LLC.

Chapter 5 is a case study showing how TCM3 can be applied to the running „Fred and Alice" scenario without a lot of the detail of TCM2. It includes comparisons of TCM3 with RBAC, TCM1, and TCM2.

Chapter 6 details a TCM3 class which can be applied to any authorisation scenario. For completeness, a TCM2 class is also detailed. The classes are specified using LinQ (1.4.4) to link to an SQL Server database.

1.8 Research Contributions

- Exploration of B as a tool for access control, with a complete re-specification of the RBAC formal model. This research was presented at a B Method conference held

at Nantes in 2008 (Dunne, et al., 2008). This conference was attended by the inventor of the B formal method: Jean-Raymond Abrial.

- A new technique for structuring the formal description of access control mechanisms which reduces overall complexity
- Examination of the mathematical basis of the TCM using B as the main tool
- The simplification of and the removal of inconsistencies from the TCM whilst keeping the power of the original model.
- The development of more general models of the TCM driven by the underlying mathematics
- The formalisation of a general model of access control which is a generalisation of Role Based Access Control and can be used in any authorisation scenario.
- The development of tools to enable the implementation of general authorisation models including the TCM.

1.9 Summary

The aims and objectives have been described. There has been some general discussion on the history of access control and the development of the TCM in particular. The tools that are to be applied, initially to RBAC and then to the TCM, have been described, and the use of new technologies, such as LinQ, have been mentioned.

2 Role-Based Access Control

2.1 Introduction

The starting point for the TCM specification was the ANSI INCITS standard for Role-Based Access Control (INCITS, 2004), and therefore it was natural for the underlying model in the RBAC standard to be specified as the first step.

However, the application of the B-Method, using the B-Toolkit development system, resulted in a new, comparatively readable, and understandable formal specification for RBAC. In addition, several improvements in the RBAC Standard were contained in the B specification, namely corrections to operations; the ensurance of consistency; simpler specification of separation of duties; and a better basis for implementation. Hence, this work forms a contribution to RBAC and its formal specification.

During this work, a new approach to structuring B specifications, which is appropriate for large applications such as RBAC, was also developed and used. Our re-specification in B is intended to provide a foundation for additional work that includes the extension of RBAC, in particular our own work on the TCM.

2.1.1 Core RBAC

Core RBAC embodies the essential aspects of RBAC. The basic concept of RBAC is that users are assigned to roles, permissions are assigned to roles, and users acquire permissions by being members of roles. Core RBAC includes requirements that user-role and permission-role assignment can be many-to-many. Thus, the same user can be assigned to many roles and a single role can have many users. Similarly, for permissions, a single permission can be assigned to many roles and a single role can be assigned to many permissions.

Core RBAC includes requirements for user-role review whereby the roles assigned to a specific user can be determined as well as the users assigned to a specific role. A similar requirement for permission-role review is imposed as an advanced review function. Core RBAC also includes the concept of user sessions, which allows selective activation and deactivation of roles. Finally, Core RBAC requires that users be able simultaneously to exercise permissions of multiple roles.

2.1.2 Hierarchical RBAC

Hierarchical RBAC adds requirements for supporting role hierarchies. A hierarchy is mathematically a partial order defining a parent/ child relation (the closure of which defines an ancestor/ descendant relationship) between roles, whereby descendant roles acquire the permissions of their ancestors. A medical example would be that the role "Health Care Professional" could be the parent of "GP", and the role "GP" could be the parent of role "Head of Practice". Thus, "Head of Practice" would be a descendant of "Health Care Professional". "GP" would acquire the permissions of "Health Care Professional" in addition to permissions of their own. "Head of Practice" would acquire the permissions of "GP" in addition to permissions of their own. There are generally two types of role hierarchies:

- Limited Hierarchies: This is a simple tree structure where a parent role can have several children roles, but every child role can have only one parent role.
- General Hierarchies: A parent role can have several children roles, and every child role can have several parent roles. Throughout this document, I have chosen to work with general hierarchies, this being consistent with the methodology of specifying the most general case possible throughout.

2.1.3 Constrained RBAC

2.1.3.1 Static Separation of Duty

In RBAC, separation of duty is used to enforce conflict of interest policies. Conflict of interest in a role-based system may arise as a result of a user gaining authorisation for permissions associated with conflicting roles. One means of preventing this form of conflict of interest is though static separation of duty (SSD), that is, to enforce constraints on the assignment of users to roles. An example of such a static constraint is the requirement that two roles be mutually exclusive; for example, if one role requests expenditures and another approves them, the organization may prohibit the same user from being assigned to both roles. The SSD policy can be centrally specified and then uniformly imposed on specific roles:

- Static Separation of Duty. SSD places constraints on the assignments of users to roles. Membership in one role may prevent the user from being a member of one or more other roles, depending on the SSD rules enforced.

- Static Separation of Duty in the Presence of a Hierarchy. This type of SSD relation works in the same way as basic SSD except that both inherited roles as well as directly assigned roles are considered when enforcing the constraints.

With respect to the constraints placed on the user-role assignments for defined sets of roles, the RBAC standard defines SSD as a pair (*roleset*, *n*) where no user is assigned to *n* or more roles from the role set.

2.1.3.2 Dynamic Separation of Duty

In RBAC dynamic separation of duty (DSD), like SSD, limits the permissions that are available to a user. However, DSD differs from SSD relations by the context in which these limitations are imposed. DSD requirements limit the availability of the permissions by placing constraints on the roles that can be activated within or across a user‟s sessions. (A user‟s activated roles for a particular session are a subset of their assigned roles.)

Similar to SSD relations, DSD defines constraints as a pair (*roleset*, *n*) with the property that no user session may activate *n* or more roles from the role set.

2.2 RBAC Reference Model

I define the RBAC reference model following the same outline as the NIST standard. However, I take the B-Method as our means of definition. This is an improvement over the Z used in the standard and exposes some of the errors. It also has the advantage of generating and requiring the discharge of proof obligations. Even disregarding the mathematical simplification provided by the B-Model, the increased validity provided by the discharge of proof obligations means that this specification is significantly better than the NIST standard, although this may not be apparent in just comparing specifications.

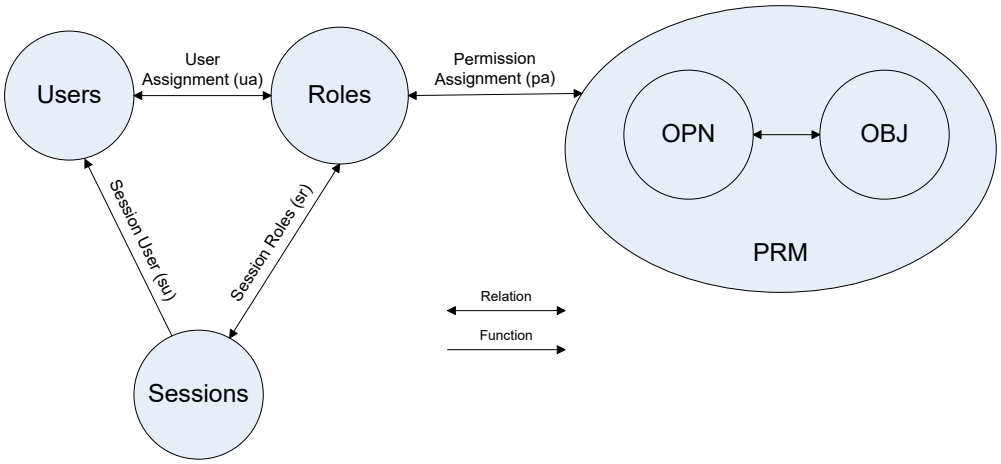


Figure 2: Core RBAC

2.2.1 Core RBAC

Core RBAC model sets and relations are shown in Figure 2. Core RBAC includes five basic set types called USER, ROLE, OPN (operation), OBJ (object), and SESSION. The set type PRM (permission) is the cross product of the types operation and object: PRM = OPN x OBJ. The RBAC standard has PRM as a separate type, but if anything, permissions would be a variable i.e. a subset of OPN x OBJ consisting of valid operations on objects. In this treatment, it has not been found necessary to have a permissions variable separate from the permission to role relation.

The RBAC model as a whole is fundamentally defined in terms of roles being assigned to users and permissions being assigned to roles. In addition, the Core RBAC model includes a set of sessions where each session is a mapped to an activated subset of the roles that have been assigned to the user.

A user is thought of as a human being. However, the concept of a user can be extended to include machines, networks, or intelligent autonomous agents. A role is a job function within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role. Permission is an approval to perform an operation on one or more RBAC protected objects. An operation is an executable image of a program, which upon invocation executes some function for the user.

The types of operations and objects that RBAC controls are dependent on the type of system in which they will be implemented. For example, within a file system, operations

might include read, write, and execute; within a database management system, operations might include insert, delete, append, and update.

2.2.2 Hierarchical RBAC

Hierarchical RBAC introduces role hierarchies through a parent/ child relationship (*pc*) as indicated in

Figure **3**. The relation *ad* defines the ancestor to descendant relationships as the reflexive transitive closure of the parent/ child relationship: $ad = pc^*$.

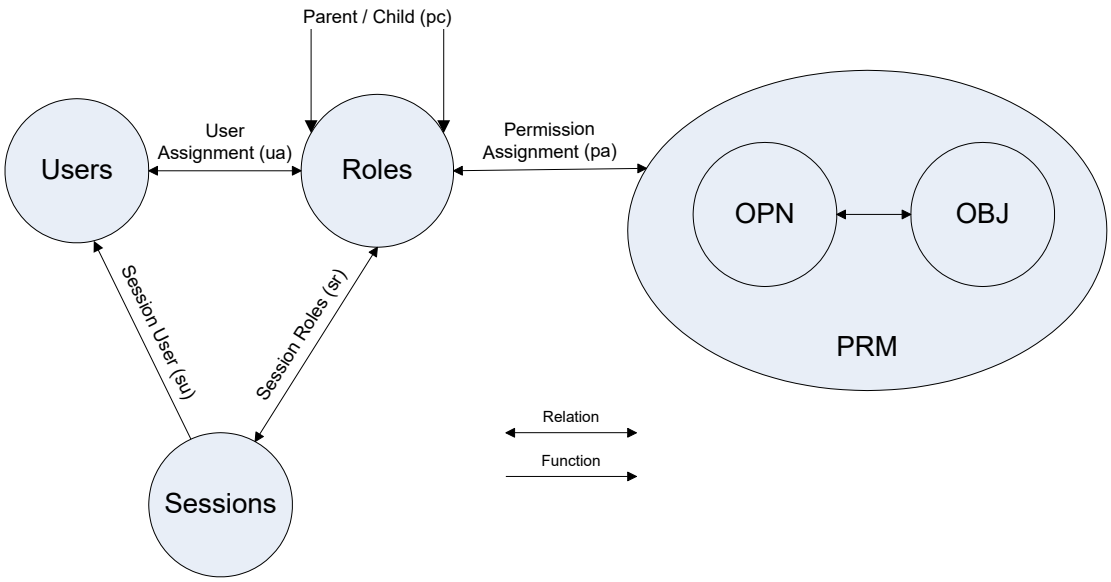


Figure 3: Hierarchical RBAC

2.2.3 Constrained RBAC

Constrained RBAC introduces two partial functions:

$$ssd \in \mathbf{F}_1\ (Roles) \nrightarrow \mathbf{N}_1$$
$$dsd \in \mathbf{F}_1\ (Roles) \nrightarrow \mathbf{N}_1$$

$\mathbf{F}_1$ (*Roles*) is the set of all non-empty finite subsets of Roles. $\mathbf{N}_1$ is the set of non-zero natural numbers. The function *ssd* defines the maximum number of members of each Roles subset that can be concurrently authorised by user assignment and role hierarchies. The function *dsd* defines the maximum number of members of each Roles subset that can be contained in the set of roles for a session.

2.3 Specification in B

The basic building block of specification in B is an abstract machine. An abstract machine is usually a specification of part of a system. A machine is not unlike a description of an object in an object-oriented sense. It has a name, some internal state, and a set of operations. Machines can be included in other machines. Figure 4 shows the machines and their inclusion structure for the modelling of the RBAC Standard. Each rectangle represents a machine, and the arrows show their inclusion structures (i.e. that a machine is specified in terms of any included machines).

Figure 4 also shows our new technique for specifying complex B applications such as RBAC. The machine *icrbac* contains the specification of essential RBAC functionality (which corresponds to Core RBAC in the RBAC Standard); this machine is then included in the *ihrbac* machine which contains the specification of hierarchical RBAC (corresponding to Hierarchical RBAC in the RBAC Standard). However the machine *crbac* just addresses the functionality of Core RBAC, for the purposes of a user who

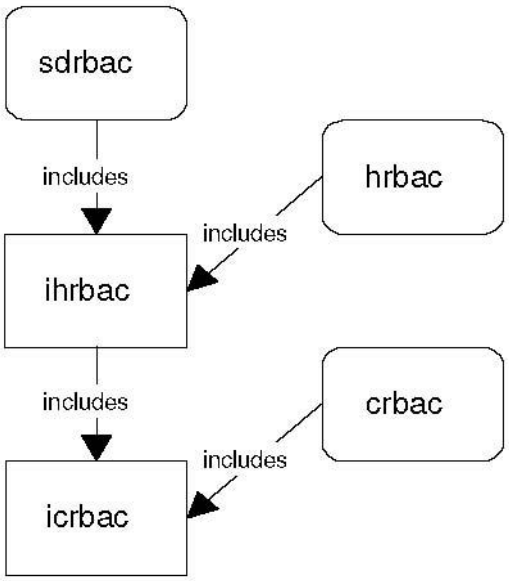


Figure 4: Abstract Machines for RBAC

just wants to use Core RBAC –It is a "shell" providing a Core RBAC interface and including those invariants and preconditions which are only applicable to Core RBAC. The machines *ihrbac* and *hrbac* perform a similar function for Hierarchical RBAC. The machine *sdrbac* is RBAC with Separation of Duties or Constrained RBAC.

The following sections consist of the actual specification of machines for RBAC, as developed using the B-Toolkit. The specification follows the layout of the B-Toolkit, which

allows comments to be included with the machine specifications. These specifications can, and have been verified, using the B-Toolkit.

2.3.1 Inner Core RBAC (*icrbac*)

This machine is the inner core of core RBAC. It consists of those variables and operations which can be included in core RBAC, or included in RBAC with general or limited hierarchies.

MACHINE

Every B machine has a name which can be used to include the machine in other machines

icrbac

SEES

Bool_ TYPE is a basic machine which adds Boolean functionality.

Bool_ TYPE

SETS

Sets introduced in this clause are types available for use in the rest of the machine. They can be named here without any further information being provided, deferring their definition until some later state of the development. There are five types declared here in contrast to the ANSI standard, which has everything of type NAME. This gives a more exact verification when type checking is performed by the B-Toolkit

USER; ROLE; OPN; OBJ; SESSION

VARIABLES

An abstract machine maintains its state information in variables. The following are the variables used in RBAC and defined in the ANSI standard.

Users, Roles, ua, pa, su, sr

INVARIANT

The invariant clause provides all of the information about the variables of the machine. It must give all of their types, and their relationship with each other. The values of variables will change as the machine executes, but the invariant describes the properties of the variables, which must be always true as the machine executes. The invariant clauses below closely follows the ANSI standard.

$Users \subseteq USER \wedge$

$Roles \subseteq ROLE \wedge$

$ua \in Users \leftrightarrow Roles \wedge$

$pa \in PRM \leftrightarrow Roles \wedge$

The variable *su* is defined as a partial function from SESSION to the set of users. The set *Sessions* is defined as the domain of *su* in **DEFINITIONS** and *UserSessions* is defined as the inverse of *su*. The variable *sr* (session roles) is a relation between Sessions and Roles.

$su \in SESSION \nrightarrow Users \wedge$

$sr \in Sessions \leftrightarrow Roles$

INITIALISATION

The initialisation clause describes the possible initial state of the machine. All variables listed in the variables clause must be assigned some value. The initialisation state must satisfy the invariant.

$Users, Roles, ua, pa, su, sr$

$:= \{\} , \{\} , \{\} , \{\} , \{\} , \{\}$

DEFINITIONS

This section defines derived variables. There is no need for them to be separately updated from the variables they derive from as is done in the RBAC standard. *PRM* is the type for permissions and is the cross product of operation (*OPN*) and object (*OBJ*):

PRM = OPN x OBJ;

The permissions (*prms*) in use i.e. the operations on objects that are assigned to all the different roles in the system are derived from the permission to role relation *pa*:

prms = **dom** (pa);

The set of *Sessions* is derived from session to user function *su*:

Sessions = **dom** (su);

The set of *sessions* belonging to the user *u*:

UserSessions (u) = $su^{-1}[\{u\}]$;

Roles active in a session *s*:

Activeroles (s) = sr [{s}];

All roles active for a user *u* in all sessions belonging to the user:

Rolesactive (u) = sr [su^{-1}[{u}];

Roles assigned to a user *u*:

Assignedroles (u) = ua [{u}];

Users assigned to a role *r*:

Assignedusers (r) = ua^{-1}[{r}];

Assigned permissions for a role *r*:

Assignedpermissions (r) = pa^{-1}[{r}];

Permissions assigned to a set of roles *s*:

Permissionsassigned (s) = pa^{-1}[s]

OPERATIONS

This clause contains a list of operation definitions. The B description of an operation provides preconditions, input parameters, output parameters, and the effects or behaviour of the operation. It is in a form suitable for software specification.

2.3.1.1 Administrative Commands

AddUser: This operation creates a new RBAC user. The user must be of type USER and not already, a member of the *Users* set. There is no need to update *UserSessions* as in the RBAC standard because *UserSessions* is derived from *su* and initially maps the user to the empty set by definition.

AddUser (user) ≙
  PRE
    user ∈ USER − Users
  THEN
    Users: = Users ∪ {user}
  END;

DeleteUser: This operation deletes an existing user from the RBAC database. The operation is valid if and only if the user to be deleted is a member of *Users*. The user assignment relation (*ua*) and the session user function (*su*) are updated. The user's sessions are removed from the session role (*sr*) relation. The maths in the RBAC standard is over complicated. The effect of *DeleteUser*, *DeleteRole*, and *DeassignUser* on sessions and session roles are left in the ANSI standard as implementation decisions. The

alternatives are sessions could be allowed to continue, forced to terminate, or disallowed roles removed from the session. Particular choices have been made in the operations below but others could be equally well implemented. In fact B allows indeterminate assignment, with precise specification at a later refinement stage.

DeleteUser (user) ≙
  PRE
    user ∈ Users
  THEN
    Users := Users – {user} ||
    ua := {user} ⩤ ua ||
    su := su ⩥ {user} ||
    sr := User_Sessions (user) ⩤ sr
  END;

AddRole: This operation creates a new RBAC role. The role must be of type ROLE and not already a member of the *Roles* set. There is no need to update user assignment and permission assignments as in the RBAC standard because these are automatically mapped to the empty set by definition.

AddRole (role) ≙
  PRE
    role ∈ ROLE - Roles
  THEN
    Roles := Roles ∪ {role}
  END;

XDeleteRole: This operation deletes an existing role from the RBAC database. The operation is valid if and only if the role to be deleted is a member of *Roles*. The user assignment relation (*ua*) and the permission assignment relation (*pa*) are updated. The role is removed from the session role (*sr*) relation. The operation is named XDeleteRole because this specification is going to be included in a higher order machine. This is the case for all operations beginning with X. The maths in the RBAC standard is vastly overcomplicated and a lot of it is unnecessary.

XDeleteRole (role) ≙
  PRE
    role ∈ Roles
  THEN
    Roles := Roles - {role} ||

ua := ua ⩤ {role} ||

pa := pa ⩤ {role} ||

sr := sr ⩤ {role} ||

END;

"*XAssignUser*: This operation assigns a user to a role. The operation is valid if and only if the user is a member of *Users*, the role is a member of *Roles*, and the user is not already assigned to the role. For the definition of *Assignedroles* see **DEFINITIONS**
There is no need to update the derived function *Assignedusers* as is done in the RBAC standard."

XAssignUser (user, role) ≙

PRE

user ∈ Users ∧

role ∈ Roles – Assignedroles (user)

THEN

ua := ua ∪ {user ↦ role}

END;

XDeassignUser: This operation deletes the assignment of the user to the role. The operation is valid if and only if the user is a member of *Users* and the role is one of the user"s assigned roles. In this case, it has been decided that a role cannot be deassigned whilst it is one of the user"s active roles. For a definition of *Rolesactive* see **DEFINITIONS**

XDeassignUser (user, role) ≙

PRE

user ∈ Users ∧

role ∈ Assignedroles (user) – Rolesactive (user)

THEN

ua := ua - {user ↦ role}

END;

GrantPermission: This operation grants a role the permission to perform an operation on an object. The permission must not already be one of the role"s Assignedpermissions. For the definition of Assignedpermissions see **DEFINITIONS**

.

GrantPermission (op, obj, role) ≙

PRE

op ∈ OPN ∧ obj ∈ OBJ ∧ role ∈ Roles ∧

op ↦ obj ∉ Assignedpermissions (role)

THEN

pa := pa ∪ {op ↦ obj ↦ role}

END;

RevokePermission: This operation revokes the permission to perform an operation on an object from the set of permissions assigned to a role. The permission must be one of the role"s Assignedpermissions. For a definition of *Assignedpermissions* see **DEFINITIONS**

.

RevokePermission (op, obj, role) ≙

PRE

op ∈ OPN ∧ obj ∈ OBJ ∧ role ∈ Roles ∧

op ↦ obj ∈ Assignedpermissions (role)

THEN

pa := pa - {op ↦ obj ↦ role}

END;

2.3.1.2 Supporting System Functions

XCreateSession: This operation creates a new session for a given user as owner with an active role set *ars*. The operation is valid if and only if *user* is a member of *Users*, and the active role set is a subset of *Roles*. The requirement that the active role set is a subset of the session user"s assigned roles is enforced through the invariant sr ⊆ (su ; ua)

in crbac. This invariant is different when „inner core rbac" is included in „hierarchical rbac". The set *Sessions* is defined as the domain of *su* in **DEFINITIONS**

XCreateSession (user, session, ars) ≙

PRE

user ∈ Users ∧ session ∈ SESSION - Sessions ∧ ars ⊆ Roles

THEN

su (session) := user ||

sr := sr ∪ session x ars

END;

DeleteSession: This operation deletes a given session. The function is valid if and only if *session* is a member of *Sessions*. The relations, session user (*su*) and session roles (*sr*) are modified using domain subtraction. There is no need to input the user as an additional parameter as is done in the NIST standard.

DeleteSession (session) ≙
PRE
session ∈ Sessions
THEN
su := {session} ⩤ su ||
sr := {session} ⩤ sr
END;

XActivateRole: This operation adds a role as an active role of a session. The requirement that role is one of the session user"s assigned roles is enforced through the invariant *sr ⊆ (su ; ua)*
in crbac. There is no reason to add user to the operation as a parameter (as done in the standard) as the session belongs to one and only one user. For a definition of *Activeroles* see **DEFINITIONS**

XActivateRole (session, role) ≙
PRE
session ∈ Sessions ∪ role ∈ Roles ∧
role ∈ Roles - Activeroles (session)
THEN
sr := sr ∪ {session ↦ role}
END;

DeactivateRole: This function deletes a role from the active role set of a session. The function is valid if and only if the session is a member of Sessions, and the role is an active role of that session. There is no reason to have user in the operation as a parameter (as done in the standard) as the session belongs to one and only one user.

DeactivateRole (session, role) ≙
PRE
Session ∈ Sessions ∪ role ∈ Roles ∧
role ∈ Activeroles (session)
THEN
sr := sr - {session ↦ role}
END;

CheckAccess: This operation returns a Boolean value as to whether the user of a given session is or is not allowed to perform a given operation on a given object. The function is

valid if and only if the session is a member of Sessions, the object is of type OBJ, and the operation is of type OPN. For a definition of *Permissionsassigned* see **DEFINITIONS**

.

bb ← **CheckAccess** (session, op, obj) ≙
PRE
session ∈ Sessions ∧ op ∈ OPN ∧ obj ∈ OBJ
THEN
bb := **bool** (op ↦ obj ∈ Permissionsassigned (Activeroles (session)))
END;

2.3.1.3 Review Functions

AssignedUsers: This function returns the set of users assigned to a given role. The function is valid if and only if the role is a member of *Roles*. For a definition of *Assignedusers* see **DEFINITIONS**

au ← **AssignedUsers** (role) ≙
PRE
role ∈ Roles
THEN
au := Assignedusers (role)
END;

AssignedRoles: This operation returns the set of roles assigned to a given user. The operation is valid if and only if the user is a member of *Users*. For a definition for Assignedroles is see **DEFINITIONS**

ar ← **AssignedRoles** (user) ≙
PRE
user ∈ Users
THEN
ar := Assignedroles (user)
END;

2.3.1.4 Advanced Review Functions

SessionRoles: This operation returns the active roles associated with a session. The operation is valid if and only if *session* is a member of *Sessions*.

sr ← **SessionRoles** (session) ≙
PRE

session ∈ Sessions
THEN
sr := Activeroles (session)
END;

"*SessionPermission*s: This operation returns the permissions of the session, that is, the permissions assigned to its active roles. The operation is valid if and only if *session* is a member *Sessions*."

sp ← **SessionPermissions** (session) ≙
PRE
session ∈ Sessions
THEN
sp := Permissionsassigned (Activeroles (session))
END

2.3.2 Core RBAC (crbac)

Core RBAC is the outer shell needed to implement RBAC without hierarchies or separation of duties. It includes Inner Core RBAC (icrbac), and has additional invariants and preconditions required for implementation. These invariants and preconditions are different for hierarchical and constrained RBAC which is why they are just part of the outer shell

MACHINE

crbac

INCLUDES

icrbac

PROMOTES

The promotes section includes the operations from an included machine without alteration:

AddUser, DeleteUser, AddRole, GrantPermission, RevokePermission, DeleteSession, DeactivateRole, CheckAccess, AssignedUsers, AssignedRoles, SessionRoles, SessionPermissions

INVARIANT

sr ⊆ (su ; ua)

This expression enforces the requirement that a session"s roles must be a subset of the session user"s assigned roles. In order to satisfy this requirement the preconditions of

some operations have to be altered. In discharging the proof obligations any preconditions that do not satisfy the invariant are highlighted, and can be altered until all the proof obligations are discharged.

The power of this procedure was not available to the producers of the RBAC standard, with the result of many errors – usually minor – in the standard. However, there is a case for saying that a standard that is going to be used as a basis for programming and development should be free of all errors even minor ones.

As an example of the type of error found, there is no special handling of „DeleteRole" for Hierarchical RBAC in the RBAC standard: it being assumed that it would be the same as in Core RBAC. However, if the deleted role is contained within the inheritance relationship, then this has to be addressed as part of role deletion (see 2.3.3.1).

It is primarily because of the discharge of the proof obligations that it is said that this specification is a better basis for programming than the standard. In fact, code can be directly produced in C from the specification using the B-Toolkit. However, because I am using SQL and C#, the code parallels the specification rather than being directly produced, but the correlation and the help given by the formal specification is clear.

OPERATIONS

2.3.2.1 Administrative Commands

DeleteRole: This operation deletes an existing role from the RBAC database and has the same preconditions as XDeleteRole. The effect in Core RBAC is that the included operation is renamed to give the final operation.

DeleteRole (role) ≙
  PRE
    role ∈ Roles
  THEN
    XDeleteRole (role)
  END;

AssignUser: This operation assigns a user to a role and has the same preconditions as XAssignUser. The effect in Core RBAC is that the included operation is renamed to give the final operation.

AssignUser (user, role) ≙
  PRE

user ∈ Users ∧

role ∈ Roles – Assignedroles (user)

THEN

XAssignUser (user, role)

END;

"*DeassignUser*: This operation deletes the assignment of the user to the role and has the same preconditions as XDeassignUser. The effect in Core RBAC is that the included operation is renamed to give the final operation"

DeassignUser (user, role) ≙

PRE

user ∈ Users ∧

role ∈ Assignedroles (user) – Rolesactive (user)

THEN

XDeassignUser (user, role)

END;

2.3.2.2 Supporting System Functions

CreateSession: This operation creates a new session for a given user as owner with an active role set (*ars*). There is an additional precondition which is required to satisfy the invariant *sr ⊆ (su; ua)*, and that is that the active role set (*ars*) must be a subset of the roles assigned to that user.

CreateSession (user, session, ars) ≙

PRE

user ∈ Users ∧ session ∈ SESSION - Sessions ∧

ars ⊆ Assignedroles (user)

THEN

XCreateSession (user, session, ars)

END;

ActivateRole: This operation adds a role as an active role of a session and has the same preconditions as XActivateRole. To satisfy the invariant *sr ⊆ (su; ua)* the *role* must be one of the session user"s assigned roles.

ActivateRole (session, role) ≙

PRE

session ∈ Sessions ∧

role ∈ Roles - Activeroles (session) ∧

role ∈ Assignedroles (su (session))
THEN
XActivateRole (session, role)
END;

2.3.2.3 Advanced Review Operations

RolePermissions: This operation returns the set of all permissions (*op*, *obj*), granted to a given role. The operation is valid if and only if the role is a member of *Roles.*

rp ← **RolePermissions** (role) ≙
PRE
role ∈ Roles
THEN
rp := AssignedPermissions (role)
END;

UserPermissions: This operation returns the permissions a given user gets through his or her assigned roles. The operation is valid if and only if the user is a member of *Users*.

up ← **UserPermissions** (user) ≙
PRE
user ∈ Users
THEN
up := Permissionsassigned (Assignedroles (user))
END;

RoleOperationsOnObject: This operation returns the set of operations a given role is permitted to perform on a given object. The operation is valid only if the role is a member of *Roles*, and the object is of type OBJ.

ro ← **RoleOperationsOnObject** (role, obj) ≙
PRE
role ∈ Roles ∧ obj ∈ OBJ
THEN
ro: = {op | op ∈ OPN ∧ op ↦ obj ∈ Assignedpermissions (role)}
END;

UserOperationsOnObject: This operation returns the set of operations a given user is permitted to perform on a given object, obtained either directly or through his or her assigned roles. The operation is valid if and only if the user is a member of *Users* and the object *obj* is of type OBJ.

uo ← **UserOperationsOnObject** (user, obj) ≙

PRE

user ∈ Users ∧ obj ∈ OBJ

THEN

ro := {op | op ∈ OPN ∧ op ↦ obj ∈ Permissionsassigned(Assignedroles (user))}

END

2.3.3 Inner Hierarchical (ihrbac)

MACHINE

This machine is the inner core of RBAC with general hierarchies. It consists of those operations which can be included in hierarchical RBAC or in a higher machine which also has separation of duties:

ihrbac

SEES

Bool TYPE

INCLUDES

Inner Hierarchical RBAC includes Inner Core RBAC:

icrbac

PROMOTES

The promotes section includes the operations from an included machine without alteration:

AddUser , DeleteUser , AddRole , XAssignUser , GrantPermission , RevokePermission , DeleteSession , DeactivateRole , CheckAccess , AssignedUsers , AssignedRoles , SessionRoles , SessionPermissions

VARIABLES

The additional variable in Hierarchical RBAC is the parent/ child relation *pc* which is used to implement General or Limited Role Hierarchies.

pc

INVARIANT

The pc relation is a mapping from *Roles* to *Roles*. The expression $r1 \mapsto r2 \in pc$ implies that *r1* is a parent of *r2* and conversely *r2* is a child of *r1*. E.g. doctor would be the parent of paediatrician if there were no intervening roles, or engineer the parent of chief engineer.

$pc \in Roles \leftrightarrow Roles$

It is necessary to avoid a cycle so that a role cannot be an ancestor of itself. The invariant below enforces that. The definition of cycles is in **DEFINITIONS**

below.

cycles (*pc*) = {}

The following invariant enforces the condition that a user"s session roles are a subset of their authorised roles either by assignment or through role hierarchies. In Core RBAC, the expression was *sr* ⊆ (*su* ; *ua*) but now we must include roles acquired through the parent/ child relation *pc*. The definition of cycles *uauth* is given in **DEFINITIONS**

.

sr ⊆ (*su* ; *uauth*)

INITIALISATION

The initialisation clause describes the possible initial state of the machine. All variables listed in the variables clause must be assigned some value. The initialisation state must satisfy the invariant.

pc := {}

DEFINITIONS

The definition of *ad* gives the ancestor to descendant relationships as the reflexive transitive closure of the parent to child relationship *pc*:

ad = pc *;

The definition of *da* gives the descendant to ancestor relationships as the inverse reflexive transitive closure of the parent to child relationship pc:

da = pc * $^{-1}$;

The descendant to ancestor relationship together with initial user assignment *ua* enables the definition of user authorisation *uauth* as below:

uauth = (ua ; pc * $^{-1}$);

The definition below gives the roles authorised to a user *u* through assignment and inheritance based on the definition of *uauth* above:

Authorisedroles (u) = ua ; pc * $^{-1}$ [{u}];

The definition below gives the users authorised to a role *r* through assignment and inheritance.

Authorisedusers (r) = pc * ; ua^{-1} [{r}];

The definition below gives the authorised permissions for a role *r*:

Authorisedpermissions (r) = (pa ; pc *) $^{-1}$ [{r}];

The definition below gives the permissions authorised to a set of roles *s:*

Permissionsauthorised (s) = (pa ; pc *) $^{-1}$ [s];

The relation cycles (*r*) identifies the cycles of a homogeneous relation *r*. It consists of maplets of the form $x \mapsto x$ where *x* is in the domain of *r* and is related directly or indirectly by *r* to itself:

cycles (r) = **id** (**dom** (r)) ∩ ∐ii. (ii ∈ N_1 | $(r)^{ii}$)

END;

OPERATIONS

2.3.3.1 Administrative Commands

XXDeleteRole: The precondition for *XXDeleteRole* is that the role is in *Roles*, and that there is no inheritance with *role* as a child i.e. any inheritance with *role* as a child must be deleted before *role* is removed. This ensures that the invariant *sr* ⊆ (*su* ; *uauth*) is not broken. The ROLE *role* is removed from permission assignment (*pa*), user assignment (*ua*) and session roles (*sr*) by *XDelete* (*role*) and from the inheritance relation by *pc* := {*role*} ⩤ *cp*.

XXDeleteRole (role) ≙
PRE
role ∈ Roles – **ran** (pc)
THEN
pc := {role} ⩤ pc ||
XDelete (role)
END;

DeassignUser: The preconditions for *DeassignUser* are the same as for *XDeassignUser* with the additional requirement that there are no sessions involving the user. This is to avoid breaking the invariant $sr \subseteq (su \; ; uauth)$ where $uauth = (ua \; ; pc^{*-1})$.

DeassignUser (user, role) ≙
PRE
user ∈ Users ∧
role ∈ AssignedRoles (user) ∧
su ▷ {user} = {}
THEN
XDeassignUser (user, role)
END;

XAddInheritance: The XAddInheritance operation establishes a parent/ child relationship between *rp* and *rc*. The precondition $(rp \mapsto rc \notin da)$ avoids cycle creation. For a definition of *da* see **DEFINITIONS**.

XAddInheritance (rp, rc) ≙
PRE
rp ∈ Roles ∧ rc ∈ Roles ∧
rp ↦ rc ∉ pc ∧
rp ↦ rc ∉ da
THEN
pc := pc ∪ {rp ↦ rc}
END;

DeleteInheritance: This command deletes an existing parent/ child inheritance maplet $rp \mapsto rc$. The command is valid if and only if the roles *rp* and *rc* are members of *Roles*. The requirement $sr \subseteq (su \; ; ua \; ; (pc - \{rp \mapsto rc\})^{*-1}$ ensures that the invariant $sr \subseteq (su \; ; uauth)$ isn‟t broken.

DeleteInheritance (rp, rc) ≙
PRE
rp ∈ Roles ∧ rc ∈ Roles ∧
rp ↦ rc ∈ pc ∧
sr ⊆ (su; ua; (pc – {rp ↦ rc})$^{*-1}$
THEN
pc := pc ∪ {rp ↦ rc}
END;

AddAscendant: This operation creates a new role *rp*, and inserts it in the role hierarchy as the parent of the existing role *rc*. The command is valid if and only if *rp* is of type ROLE but not a member of *Roles* and *rc* is a member of *Roles*.

AddAscendant (rp, rc) ≙
PRE
rp ∈ ROLE - Roles ∧ rc ∈ Roles
THEN
AddRole (rp) ||
pc := pc ∪ {rp ↦ rc}
END;

AddDescendant: This operation creates a new role *rc*, and inserts it in the role hierarchy as the child of the existing role *rp*. The command is valid if and only if *rc* is of type ROLE but not a member of *Roles* and *rp* is a member of *Roles*.

AddDescendant (rp, rc) ≙
PRE
rp ∈ Roles ∧ rc ∈ ROLE - Roles
THEN
AddRole (rc) ||
pc := pc ∪ {rp ↦ rc}
END;

2.3.3.2 Supporting System Operations

XXCreateSession: This operation creates a new session with a given user as owner and a given set of active roles. The operation is valid if and only if *user* is a member of *Users*, and the active role set is a subset of the roles authorised for that *user*. In Hierarchical RBAC, authorised roles are those assigned to the users and also those allowed from that assignment through inheritance. For a definition of *Authorisedroles* see **DEFINITIONS**.

XXCreateSession (user, session, ars) ≙
PRE
user ∈ Users ∧ session ∈ SESSION - Sessions ∧
ars ⊆ Authorisedroles (user)
THEN
XCreateSession (user, session, ars)
END;

XXActivateRole: This operation adds a role as an active role of a session. The preconditions are the same as Core RBAC except that *role* can be a member of the user‟s authorised roles which includes roles gained through inheritance, not just the user‟s assigned roles.

XXActivateRole (session, role) ≙
PRE
session ∈ Sessions ∧
role ∈ Roles - Activeroles (session) ∧
role ∈ Authorisedroles (su (session))
THEN
XActivateRole (session, role)
END;

2.3.3.3 Review Operations

AuthorisedUsers: This operation returns the set of users authorised for a given role, either by assignment or through inheritance. The operation is valid if and only if the given role is a member of *Roles*. For a definition of *Authorisedusers* see **DEFINITIONS**.

au ← **AuthorisedUsers** (role) ≙
PRE
role ∈ Roles
THEN
au := Authorisedusers (role)
END;

AuthorisedRoles: This operation returns the set of roles authorised for a given user. The operation is valid if and only if the user is a member of *Users*. For a definition of *Authorisedroles* see **DEFINITIONS**.

ar ← **AuthorisedRoles** (user) ≙
PRE
user ∈ Users
THEN
ar := Authorisedroles (user)
END;

2.3.3.4 Advanced Review Operations for General Role Hierarchies

RolePermissions: This operation returns the set of all permissions (*op, obj*), granted to the given role, or inherited from the given role‟s ancestor roles. The operation is valid if and only if the role is a member of the *Roles*. For a definition of *Authorisedpermissions* see **DEFINITIONS**.

rp ← **RolePermissions** (role) ≙
PRE
role ∈ Roles
THEN
rp := Authorisedpermissions (role)
END;

UserPermissions: This operation returns the set of permissions a given user gets through his or her authorised roles. The operation is valid if and only if the user is a member of *Users*. For a definition of *Authorisedroles* and *Permissionsauthorised* see **DEFINITIONS**.

up ← **UserPermissions** (user) ≙
PRE
user ∈ Users
THEN
up := Permissionsauthorised (Authorisedroles (user))
END;

RoleOperationsOnObject: This operation returns the set of operations a given role is permitted to perform on a given object. The set contains all operations granted directly to that role or inherited by that role from ancestor roles. The operation is valid only if the role is a member of *Roles*, and the object is of type OBJ.

ro ← **RoleOperationsOnObject** (role, obj) ≙
PRE
role ∈ Roles ∧ obj ∈ OBJ
THEN
ro := {op | op ∈ OPN ∧ op ↦ obj ∈ Authorisedpermissions (role)}
END;

UserOperationsOnObject: This operation returns the set of operations a given user is permitted to perform on a given object. The set consists of all the operations obtained by the user, either directly or through their authorised roles. The operation is valid if and only if the user is a member of the *Users* and the object is of type OBJ.

uo ← **UserOperationsOnObject** (user, obj) ≙
PRE
user ∈ Users ∧ obj ∈ OBJ
THEN
ro := {op | op ∈ OPN ∧ op ↦ obj ∈ Permissionsauthorised (Authorisedroles(user))
END

2.3.4 Hierarchical (hrbac)

MACHINE

This machine is the outer layer of RBAC with General Hierarchies. It includes inner hierarchical RBAC (*ihrbac*) which in turn includes inner core rbac (*icrbac*).

hrbac

SEES

Bool TYPE

INCLUDES

ihrbac

PROMOTES

AddUser, DeleteUser, AddRole, DeassignUser, GrantPermission, RevokePermission, DeleteInheritance, AddAscendant, AddDescendant, DeleteSession, DeactivateRole, CheckAccess, AssignedUsers, AssignedRoles, AuthorisedUsers, AuthorisedRoles, RolePermissions, UserPermissions, SessionRoles, SessionPermissions, RoleOperationsOnObject, UserOperationsOnObject

OPERATIONS

2.3.4.1 Administrative Commands

DeleteRole: This operation deletes an existing role from the RBAC database and has the same preconditions as *XXDeleteRole*. The effect in Hierarchical RBAC is that the included operation is renamed to give the final operation.

DeleteRole (role) ≙
  PRE
    role ∈ Roles – **ran** (pc)
  THEN
    XXDeleteRole (role)
  END;

AssignUser: This operation assigns a user to a role and has the same preconditions as XAssignUser. The effect in Hierarchical RBAC is that the included operation is renamed to give the final operation.

AssignUser (user, role) ≙
  PRE
    user ∈ Users ∧
    role ∈ Roles – Assignedroles (user)

THEN
XAssignUser (user, role)
END;

AddInheritance: The AddInheritance operation establishes an immediate pair relationship between *rp* and *rc* and has the same preconditions as *XAddInheritance*. The effect in Hierarchical RBAC is that the included operation is renamed to give the final operation.

AddInheritance (rp, rc) ≙
PRE
rp ∈ Roles ∧ rc ∈ Roles ∧
rp ↦ rc ∉ pc ∧
rp ↦ rc ∉ da
THEN
XAddInheritance (rp, rc)
END;

2.3.4.2 Supporting System Operations

CreateSession: This operation creates a new session with a given user as owner and a given set of active roles and has the same preconditions as *XXCreateSession.* The effect in Hierarchical RBAC is that the included operation is renamed to give the final operation.

CreateSession (user, session, ars) ≙
PRE
user ∈ Users ∧ session ∈ SESSION - Sessions ∧
ars ⊆ Authorisedroles (user)
THEN
XXCreateSession (user, session, ars)
END;

ActivateRole: This operation adds a role as an active role of a session, and has the same preconditions as *XX ActivateRole.* The effect in Hierarchical RBAC is that the included operation is renamed to give the final operation.

ActivateRole (session, role) ≙
PRE
session ∈ Sessions ∧
role ∈ Roles - Activeroles (session) ∧
role ∈ Authorisedroles (su (session))
THEN
XXActivateRole (session, role)
END;

2.3.5 Separation of Duties (sdrbac)

MACHINE

This machine is the outer layer of the onion. It includes inner hierarchical RBAC (*ihrbac*) which in turn includes inner core RBAC (*icrbac*). This machine implements all of the features of RBAC contained in the NIST standard.

sdrbac

INCLUDES

ihrbac

PROMOTES

AddUser, DeleteUser, AddRole, DeassignUser, GrantPermission, RevokePermission, DeleteInheritance, AddAscendant, AddDescendant, DeleteSession, DeactivateRole, CheckAccess, AssignedUsers, AssignedRoles, AuthorisedUsers, AuthorisedRoles, RolePermissions, UserPermissions, SessionRoles, SessionPermissions, RoleOperationsOnObject, UserOperationsOnObject

VARIABLES

ssd, dsd

INVARIANT

A partial function *ssd* (static separation of duties) is defined from the set of all non-empty finite subsets of *Roles* to the set of non-zero natural numbers. This function defines the maximum number of members of each *Roles* subset that can be concurrently authorised by user assignment and role hierarchies. The invariant ensures that the stipulated number of roles must be less than the cardinality of the roleset and that every user and authorised roleset satisfies the requirement given by the *ssd* function.

$ssd \in \mathbf{F_1}(Roles) \nrightarrow \mathbf{N_1} \wedge$
$\forall\ rs.\ (rs \in \mathbf{dom}\ (ssd) \Rightarrow ssd\ (rs) < \mathbf{card}\ (rs)) \wedge$
$\forall\ (user, rs).\ (\text{user} \in Users \wedge rs \in \mathbf{dom}\ (\text{ssd}) \Rightarrow \mathbf{card}\ (\{user\} \lhd uauth \rhd rs) \leq ssd\ (rs)) \wedge$

A partial function *dsd* (dynamic separation of duties) is defined from the set of all non-empty finite subsets of Roles to the set of non-zero natural numbers. This function defines the maximum number of members of each *Roles* subset that can be contained in the set of authorised roles for a session. The invariant ensures that the stipulated number of roles must be less than the cardinality of the roleset and that every user and authorised roleset satisfies the requirement given by the d*sd* function.

$dsd \in \mathbf{F_1}\ (Roles) \nrightarrow \mathbf{N_1} \wedge$

$\forall\ rs.\ (rs \in \text{dom}\ (dsd) \Rightarrow\ dsd\ (rs) < \mathbf{card}\ (rs))\ \wedge$

$\forall\ (session, rs).\ (session \in Sessions\ \wedge\ rs \in \mathbf{dom}\ (dsd) \Rightarrow\ \mathbf{card}\ (\{session\}\ \lhd\ sr \rhd\ \text{rs})$
$\leq\ dsd\ (rs))$

INITIALISATION

ssd, dsd := {},{}

OPERATIONS

2.3.5.1 Administrative operations

DeleteRole: This operation deletes an existing role from the RBAC database and has the same preconditions as *XXDeleteRole* with the additional requirements that *role* does not exist in any of the *ssd* or *dsd* rolesets i.e. *role* must be removed from these rolesets before the operation is carried out.

DeleteRole (role) ≙
PRE
role ∈ Roles – **ran** (pc) ∧
¬ (∃ rs. (rs ∈ dom (ssd) ∧ role ∈ rs)) ∧
¬ (∃ rs. (rs ∈ dom (dsd) ∧ role ∈ rs))
THEN
XXDeleteRole (role)
END;

AssignUser: This operation requires an SSD precondition that the new set of authorised roles for the user after adding a new inheritance does not contain more than the allowed number of members of any SSD roleset.

AssignUser (user, role) ≙
PRE
user ∈ Users ∧
role ∈ Roles – Assignedroles (user) ∧
∀ rs. (rs ∈ **dom** (ssd) ⇒ **card** ({user} ◁ (ua ∪ {user ↦ role} ; da) ▷ rs) ≤ ssd (rs))
THEN
XAssignUser (user, role)
END;

AddInheritance: This operation requires an SSD precondition that the new set of authorised

roles for any user after adding a new inheritance does not contain more than the allowed number of members of any SSD roleset.

AddInheritance (rp, rc) $\triangleq$
PRE
rp $\in$ Roles $\wedge$ rc $\in$ Roles $\wedge$
rp $\mapsto$ rc $\notin$ pc $\wedge$
rp $\mapsto$ rc $\notin$ da $\wedge$
$\forall$ (user, rs). (user $\in$ Users $\wedge$ rs $\in$ **dom** (ssd) $\Rightarrow$
card ({user} $\lhd$ ua ; (pc $\cup$ {rp $\mapsto$ rc})$^{*-1}$) $\rhd$ rs) $\leq$ ssd (rs))
THEN
XAddInheritance (rp, rc)
END;

2.3.5.2 Administrative operations for SSD Relations

CreateSsdSet: This operation takes a set of roles *rs* and sets the associated maximum number *nn* of its roles that any user can be authorised to using the partial function *ssd*. The operation is valid if and only if *rs* is a non-empty subset of Roles; and *nn* is a natural number less than the cardinality of the roleset *rs*. Additionally, no *user* in *Users* must break the ssd requirement for their currently authorised roles. If they did, this would have to be adjusted as part of creating the SSD set.

If we compare this approach to that in the NIST standard we can see that this is simpler and more complete. Also for some reason the NIST standard defines the function by saying that no user can be assigned to *nn* or more roles and that *nn* $\geq$ 2. I prefer to say that *nn* is the maximum (not *nn-1*) and that *nn* is a natural number which by definition is $\geq$ 1.

CreateSSDSet (rs, nn) $\triangleq$
PRE
rs $\in$ $\mathbf{F_1}$ (Roles) $\wedge$ nn $\in$ $\mathbf{N_1}$ $\wedge$
rs $\notin$ **dom** (ssd) $\wedge$ nn < **card** (rs) $\wedge$
$\forall$ user. (user $\in$ Users $\Rightarrow$ **card** ({user} $\lhd$ uauth $\rhd$ rs) $\leq$ nn)
THEN
ssd (rs) := nn
END;

DeleteSsdSet: This operation deletes an SSD maplet from the partial function *ssd.* It is valid if and only if the roleset *rs* is already a member of the domain of *ssd*.

DeleteSSDSet (rs) $\triangleq$
PRE

rs ∈ **dom** (ssd)
THEN
ssd := {rs} ⩤ ssd
END;

2.3.5.3 Administrative operations for DSD Relations

CreateDsdSet: This operation takes a set of roles *rs* and sets the associated maximum number *nn* of its roles that any user can have in a session using the partial function *dsd*. The operation is valid if and only if *rs* is a non-empty subset of Roles; and *nn* is a natural number less than the cardinality of the *rs*. Additionally that every *user* in *Users* must not break the DSD requirement for their currently held session roles. If they did, this would have to be adjusted as part of creating the DSD set.

CreateDSDSet (rs, nn) ≙
PRE
rs ∈ $\mathbf{F_1}$(Roles) ∧ nn ∈ $\mathbb{N}_1$ ∧
rs ∉ **dom** (dsd) ∧ nn < **card** (rs) ∧
∀ session. (session.∈ Session ⇒ **card** (sr ▷ rs) ≤ nn)
THEN
dsd (rs) := nn
END;

DeleteDsdSet: This operation deletes a DSD maplet from the partial function *dsd.* It is valid if and only if the roleset *rs* is already a member of the domain of *dsd*.

DeleteDSDSet (rs) ≙
PRE
rs ∈ **dom** (dsd)
THEN
dsd := {rs} ⩤ dsd
END;

2.3.5.4 Supporting System Operations

CreateSession: this operation requires an additional DSD precondition that the set of session roles *ars* does not contain more than the allowed number of members of any DSD roleset.

CreateSession (user, session, ars) ≙
PRE
user ∈ Users ∧ session ∈ SESSION - Sessions ∧

ars ⊆ Authorisedroles (user) ∧

∀ rs. (rs ∈ **dom** (ssd) ⇒ **card** (rs ∩ ars) ≤ dsd (rs))

THEN

XXCreateSession (user, session, ars)

END;

ActivateRole: This operation requires an additional DSD precondition that the new set of session roles after a role activation does not contain more than the allowed number of members of any DSD roleset.

ActivateRole (session, role) ≙

PRE

session ∈ Sessions ∧

role ∈ Roles - Activeroles (session) ∧

role ∈ Authorisedroles (su (session)) ∧

∀ rs. (rs ∈ **dom** (ssd) ⇒ **card** (rs ∩ (Activeroles (session) ∪ {role})) ≤ dsd (rs))

THEN

XXActivateRole (session, role)

END;

2.3.5.5 Review Operations for SSD

SsdRoleSets: This operation returns the set of all SSD rolesets.

srs ← **SsdRoleSets** (rs) ≙ srs := **dom** (ssd)

SsdRoleSetRoles: This operation returns the set of roles of a SSD role set. The operation is valid if and only if the roleset *rs* exists in the domain of *ssd*.

srsr ← **SsdRoleSetRoles** (rs) ≙

PRE

rs ∈ **dom** (ssd)

THEN

srsr := rs

END;

SsdRoleSetCardinality: This operation returns the natural number associated with a role set *rs*. The operation is valid if and only if the role set *rs* exists in the domain of *ssd*.

srsc ← **SsdRoleSetCardinality** (rs) ≙

PRE

rs ∈ **dom** (ssd)

THEN

srsc := ssd (rs)

END;

2.3.5.6 Review Operations for DSD

DsdRoleSets: This operation returns the set of all DSD role sets.

drs ← **DsdRoleSets** (rs) ≙ drs := **dom** (dsd)

DsdRoleSetRoles: This operation returns the set of roles of a DSD role set. The operation is valid if and only if the role set *rs* exists in the domain of *dsd*.

drsr ← **DsdRoleSetRoles** (rs) ≙
PRE
rs ∈ **dom** (dsd)
THEN
srsr := rs
END;

DsdRoleSetCardinality: This operation returns the natural number associated with a role set *rs*. The operation is valid if and only if the role set *rs* exists in the domain of *dsd*.

drsc ← **DsdRoleSetCardinality** (rs) ≙
PRE
rs ∈ **dom** (dsd)
THEN
srsc := dsd (rs)
END;

2.3.6 Proof Obligations

A B specification of a system of machines is a mathematical construct and precisely defines the state properties of the machines and their operations. There has to be certain properties of consistency i.e. initialisation must establish the invariants, and operations must preserve the invariants both within and between machines. These properties are called proof obligations, and the B-Toolkit can build them using a proof obligation generator. These obligations can be discharged using the B-Toolkit‟s autoprover. Failure to discharge the obligations is either because the rules used by the prover are not subtle enough, or because the machine definition contains mistakes or omissions. E.g. the prover would point out that deleting a role in Hierarchical RBAC changes a user‟s authorised permissions and thus their permitted session roles.

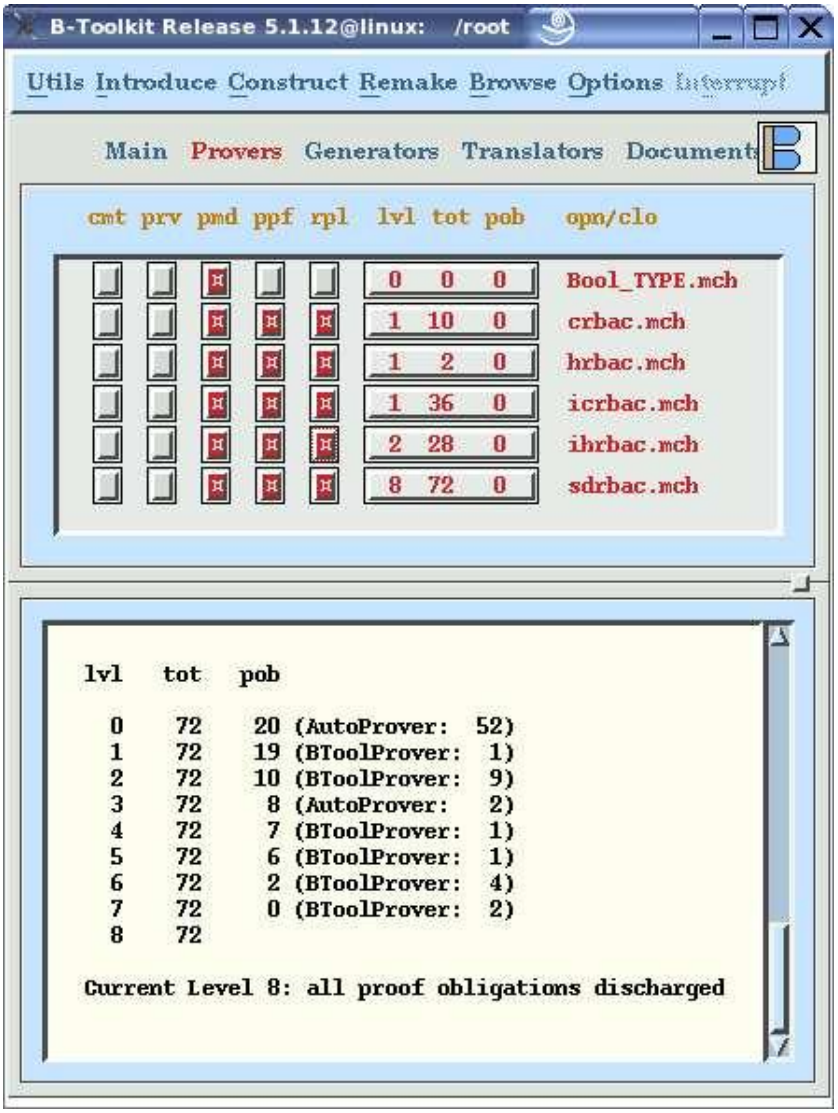


Figure 5: B-Toolkit Prover

Core RBAC specification generated only 46 proof obligations that were relatively easy to discharge. As even a simple system specification can generate many more proof obligations this is a good indication that RBAC is ideally suited to specification with the B-Method. Hierarchical RBAC generated an additional 30 proof obligations, again a low total. The increased complexity of RBAC with separation of duties generated a further 72 proof obligations. Figure 5 shows that all 148 proof obligations have been discharged.

2.4 Application of Spec# to RBAC

The following is some C# code with the addition of the Spec# superset. The main method calls other methods that interact with an SQL database. Each of the methods has pre-conditions and some have post-conditions. The methods (operations) closely follow the B

specification. This work is still very experimental. It is included here as an example of a possible way ahead in which formal methods can be included in mainstream programming to give robust code. If the code is examined it can be seen how it closely correlates to the B.

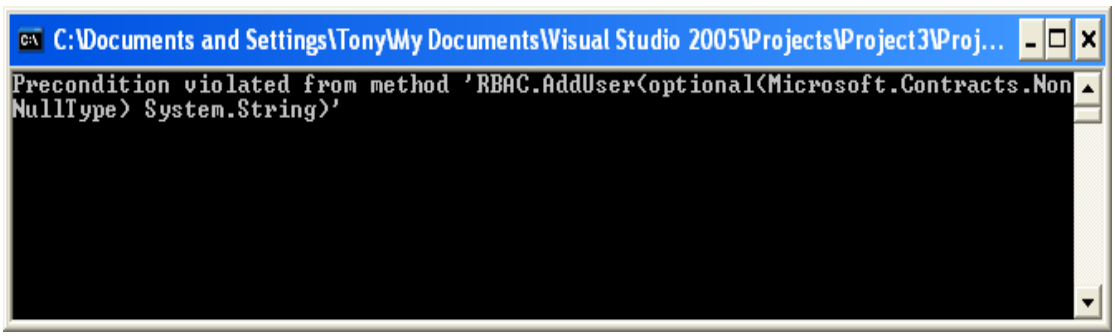
```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SpecSharp;
using Microsoft.Contracts;


class TestRBAC
{
   public static void Main()
   {
    try
     {
   string cs = "workstation id=COMP;
   packet size=4096;integrated security=SSPI;
   data source=COMP;persist security info=False;
   initial catalog=RBACdb";
   RBAC RBAC1 = new RBAC("Users","Roles",cs);
   RBAC1.AddUser("JimmySmith");
   RBAC1.AddUser("AdamWatson");
   RBAC1.AddRole("GP");
   RBAC1.AddRole("Doctor");
   RBAC1.AddRole("Psychiatrist");
   RBAC1.AddRole("Healthworker");
   RBAC1.AddInheritance("Doctor","Psychiatrist");
   RBAC1.AddInheritance("Healthworker","Psychiatrist");
   RBAC1.AssignUser("JimmySmith","Healthworker");
   RBAC1.AssignUser("JimmySmith","GP");
   RBAC1.AssignUser("AdamWatson","Psychiatrist");
   RBAC1.AssignUser("AdamWatson","GP");
   StringCollection ars = new StringCollection();
   ars.Add("GP"); ars.Add("Healthworker");
   RBAC1.CreateSession("JimmySmith","S0000001",ars);
   ars.Clear();
   ars.Add("GP"); ars.Add("Psychiatrist");
   RBAC1.CreateSession("AdamWatson","S00000226",ars);
   Console.ReadLine();
     }
   catch(Microsoft.Contracts.RequiresException mcre)
     {
       Console.WriteLine(mcre.Message);
       Console.Read();
     }
   catch(Microsoft.Contracts.EnsuresException mcee)
     {
       Console.WriteLine(mcee.Message);
       Console.Read();
     }
   }
}
```

```
class RBAC
{
   private string! Roles;
   private string! Users;
   private SqlCommand cmd;
   private SqlConnection cn;
   private DataSet dsCycles;



   public RBAC(string! u, string! r, string! cs)
   {
   Users = u;
   Roles = r;
   cn = new SqlConnection(cs);
   base();
   cmd = cn.CreateCommand();
   dsCycles = new DataSet();
   }

   public void AddUser(string! user)
   requires CountUser(user) == 0;
   {
   cmd.CommandText = "insert into " + Users + " values ('" + user + "')";
   cn.Open();
   cmd.ExecuteNonQuery();
   cn.Close();
   }

   public void AddRole(string! role)
   requires CountRole(role) == 0;
   {
   cmd.CommandText = "insert into " + Roles + " values ('" + role + "')";
   cn.Open();
   cmd.ExecuteNonQuery();
   cn.Close();
   }

   public void AssignUser(string! user, string! role)
   requires CountUser(user) == 1;
   requires CountRole(role) == 1;
   requires CountUserAssignment(user,role) == 0;
   {
   cmd.CommandText = "insert into ua values ('" +user+ "','" +role+ "')";
   cn.Open();
   cmd.ExecuteNonQuery();
   cn.Close();
   }

   public void AddInheritance(string! rp, string! rc)
   requires CountRole(rp) != 0;
   ensures Cycles() == 0;
   {
   cmd.CommandText = "insert into pc values ('" +rp+ "','" +rc+ "')";
   cn.Open();
   cmd.ExecuteNonQuery();
   cn.Close();
   }

   public void CreateSession(string! user, string session,
   StringCollection! ars)
```

```
requires CountUser(user)==1;
requires forall{int i in (0:ars.Count); CountRole(ars[i])==1 &

CountUserAuthorisation(user, ars[i])==1};
{
cn.Open();
cmd.CommandText = "insert [session_user] values ('" +session+ "','"
+user+ "')";
cmd.ExecuteNonQuery();

for(int i=0; i < ars.Count; i++)
  {
cmd.CommandText = "insert session_roles values ('" +session+ "','"
+ars[i]+ "')";
cmd.ExecuteNonQuery();
  }
cn.Close();
}

[Microsoft.Contracts.Confined]
public Int32 CountUser(string user)
{
cmd.CommandText = "select count(*) from "  + Users + " where [User] =
'" + user + "'";
cn.Open();
Int32 cu  =  Convert.ToInt32(cmd.ExecuteScalar());
cn.Close();
return cu;
}

[Microsoft.Contracts.Confined]
public Int32 CountRole(string role)
{
cmd.CommandText = "select count(*) from "  + Roles + " where Role = '"
+ role + "'";
cn.Open();
Int32 cr  =  Convert.ToInt32(cmd.ExecuteScalar());
cn.Close();
return cr;
}

[Microsoft.Contracts.Confined]
public Int32 CountSession(string session)
{
cmd.CommandText = "select count(*) from  Sessions where Session = '" +
session + "'";
cn.Open();
Int32 cr  =  Convert.ToInt32(cmd.ExecuteScalar());
cn.Close();
return cr;
}

[Microsoft.Contracts.Confined]
public Int32 CountUserAssignment(string user, string role)
{
cmd.CommandText = "select count(*) from ua where [User] = '" + user +
"'  and Role = '" + role + "'";
cn.Open();
Int32 cua  =  Convert.ToInt32(cmd.ExecuteScalar());
cn.Close();
return cua;
```

```
    }

    [Microsoft.Contracts.Confined]
    public Int32 CountUserAuthorisation(string user, string role)
    {
    cmd.CommandText = "select count(*) from uauth() where [User] = '" +
    user + "' and Role ='" + role + "'";
    cn.Open();
    Int32 cua  =  Convert.ToInt32(cmd.ExecuteScalar());
    cn.Close();
    return cua;
    }

    [Microsoft.Contracts.Confined]
    public Int32 Cycles()
    {
    cmd.CommandText = "select count(*) from cycles()";
    cn.Open();
    Int32 cc  =  Convert.ToInt32(cmd.ExecuteScalar());
    cn.Close();
    return cc;
    }
}
```

Violating a precondition or postcondition brings up a message e.g.



Console 1: Error Message

Errors can then be handled as appropriate for the application.

2.5 Summary

The RBAC NIST standard was completely specified in B. During the specification process and the discharge of the proof obligations, numerous errors were discovered in the standard. The B-Toolkit was instrumental in producing a more robust RBAC specification without those errors, which is suitable as a basis for modelling extensions and variations of RBAC. The RBAC specification produced is used as a starting point for the TCM specification in the next chapter. A small amount of Spec# has been introduced to explore the correlation between the model and the implementation code.

3 The Tees Confidentiality Model (TCM2)

The Tees Confidentiality Model (Longstaff, et al., 2003) (Longstaff, et al., 2006) is a powerful model for authorisation. It has been developed at the University of Teesside over a period of ten years. TCM1 was the application of formal methods to the Tees Confidentiality Model exactly as represented in the various papers produced up to the start of this work (Longstaff, et al., 2003) (Longstaff, et al., 2002). During the course of this work it was realised that it was possible to simplify the fundamental model without losing the power of the original. This simplified model became TCM2.

Also it was realised that although the TCM had been developed essentially as a piece of practical work to address a practical problem, it was in fact something more. It was essentially a generalisation of RBAC that also enabled authorisation to be looked at afresh from first principles.

3.1 Introduction

This chapter concerns the formal specification of the TCM. The approach mirrors the RBAC specification. The main sections of core, hierarchical and separation of duties are exactly as in RBAC, and the chapter is divided into reference model and functional specification, also as in RBAC. Some of the operations are the same as in RBAC, whilst others can be seen to be simple generalisations of RBAC operations. The TCM as currently applied, and as described in many papers, is completely specified.

3.1.1 Motivating Health Care Scenario

The health service in England is developing a national IT computer service which includes a central database (the Care Records Service, or CRS) to hold and provide continuous access to summary electronic heath records for over 50 million people (Department of Health, 2005). The original TCM has significantly contributed to this programme in that the TCM was implemented by several suppliers to the NHS as part of the Electronic Research and Development Programme (ERDIP), and it heavily influenced the CRS requirements specification (NHS, 2003). Henceforward I shall generally refer to an Electronic Health Records or EHR application, which is based on the CRS concept.

I now describe an EHR scenario for use as an example. The first part of the scenario was written by a Consultant Transplant Surgeon (Mr Michael Thick), and was used in the early TCM researches. It concerns a fictitious patient who will be referred to as Alice, and her

GP, who we will call Fred. Alice is 50; some of the major events in Alice‟s medical history are summarised as follows:

- She had a pregnancy termination when she was 16
- Was diagnosed diabetic at 25
- End Stage renal failure when she was 45
- Renal transplant at 48
- Acutely psychotic at 49
- Crush fracture of T12 aged 50

Let us now suppose, not unreasonably, that Alice expresses the desire to place the following confidentiality restrictions on the availability of her medical records data about two of these conditions (i.e. she wishes to place them in a patient‟s Sealed Envelope, in CRS terminology):

1. My GP, Fred, can see all my data.
2. Nobody must know about my termination except my GP, any Gynaecological Consultant, and the Consultant Renal Transplant Surgeon who operated on me.
3. My GP, Consultant Renal Transplant Surgeon, and Consultant Orthopaedic Surgeon can see my psychosis data, but no one else.

Let us add the following contrived requirement (but still one that a health records authorisation system must be capable of implementing):

4. I do not wish the members of the hospital team who carried out my termination operation to ever be able to see my psychosis data, except if they are viewing in a psychiatric role.

In one of the TCM demonstrators, these confidentiality requirements can be specified using electronic consent forms (Longstaff, et al., 2002). We also include the major CRS requirement that Health Care Practitioners (HCPs) will be entitled to access data based on their role, and a „Legitimate Relationship“ with the patient, generally meaning that the patient is registered with them, or has been referred to them.

This scenario is relevant to medical records access in static primary care and secondary care environments. It is also relevant to distributed processing, in which summary records may be stored on central systems, and more detailed data on other systems (this is the approach for CRS development).

In addition, we could envisage future situations where access is granted to GPs who need to access medical records from mobile devices, during home visits to patients. Portions of medical records may be downloaded, subjected to access controls, updated, and then merged back to the central database. GPS tracking could be used by to locate and assign GPs for patient home visits.

3.2 TCM2 Reference Model

3.2.1 Basic Concepts

The TCM Reference Model defines the scope and concepts of the TCM, and gives a precise, unambiguous terminology for defining the TCM Functional Specification (3.3)

3.2.1.1 RBAC Basis

The TCM can be applied within both simple and complex organizations. Therefore, following the presentation of the RBAC model in (Ferraiola, et al., 2001) (INCITS, 2004), I describe the TCM in terms of component models which I call Core, Hierarchical, Constrained and Extended; these components have use in different applications according to their increasing sophistication and complexity.

The TCM is based on the RBAC concepts of users, operations, and objects. The purpose of the TCM is to permit or deny access for a user to an operation on an object.

A user is usually defined as a human being, but generally includes machines, networks, and autonomous agents. The users of the EHR[1] system will be Health Care Professionals (including doctors, GPs, nurses), patients, and administrators. We write these as EHRsys_users.

An operation is an executable image of a program, which upon invocation executes some function for the user. The types of operations and objects that the TCM controls are dependent on the type of system in which they will be implemented. For example, within a file system, operations might include read, write, and execute; within a database management system, operations might include insert, delete, append, and update. More specifically, an EHR system might have the operations EHRsys_op_query, EHRsys_op_query_medication, EHRsys_op_add_treatment, etc. defined for EHRs.

Generally speaking, an object is an entity that contains or receives information. The TCM works with protected objects in that authorisation controls apply to such objects; we shall

[1] Electronic Health Record

henceforward simply refer to objects for brevity. Objects can represent information containers and also system resources. A patient‟s medical record maintained by an EHR system, an example of which is EHRsys_obj_Alice, is the object, together with its constituent sub-objects, that is used for illustrative purposes.

3.2.1.2 TCM2 New Concepts

In addition to the RBAC concepts described above, the TCM introduces additional concepts. Firstly, it is the concept of classifier and its associated classifier values (sometimes abbreviated to „cvalues“), which is used to specify authorisation. In the TCM, „Role“ would be an example of a user classifier, and „Engineer“ a classifier value for Role. The ordered pair <Role, Engineer> is denoted as a cvalue.

Another user classifier is EHR_users, with values having associations to users (EHRsys_users), thereby identifying collections of users. Classifiers are also specified for operations (e.g. EHR_ops, EHR_op_type) and objects (EHR_objs, EHR_obj_type). If authorisation is to be granted or denied to individual sub objects within an EHRsys_obj complex object, then this can be achieved using the classifier EHR_objs. (This might be required for denying access to any EHR data which would suggest a particular health condition, which might be found in many places in an EHRsys_obj object, e.g. sub-objects containing clinic appointments, medications, as well as diagnoses.)

A classifier quite often corresponds to a property of a user, operation, or object. However, classifier values can be any value available in the system that is useful for authorisation. There is a proposal that the concept of Legitimate Relationship be made central to authorisation in the NHS, with dedicated servers providing this information, as it exists between system users and patients. In this the classifier *LegRel*, for „Legitimate Relationship“ is derived from both user (e.g. an individual GP) and object (a patient‟s EHR). In distributed systems, the classifier values can be tokens or derived values passed from one computer to another.

Permissions in the TCM are called confidentiality permissions (abbreviated to CPs), and the types for CPs are called confidentiality permission types (CPTs). This is a development of the RBAC model, which is essentially a single confidentiality permission type. A CPT consists of a set of classifiers, for example:

CPT1= {Role, EHR_op_type, EHR_objs}.

A CP consists of a set of classifier values e.g.

CP1= {<Role, GP>, <EHR_op_type, EHR_query_medication>, <EHR_objs, EHR_Alice}, together with a mapping to a value of Permit or Deny. (Here EHR_Alice is a value associated with Alice‟s complete EHR, i.e. EHRsys_obj_Alice). The determination of an authorisation outcome is made when the classifier values of a CP match the user classifier values activated for the session together with the classifier values assigned to the operation and object being used. Therefore, the CP example above would permit GP Fred (and other GPs) to query Alice‟s medical record. These concepts are shown in Figure 6 below.

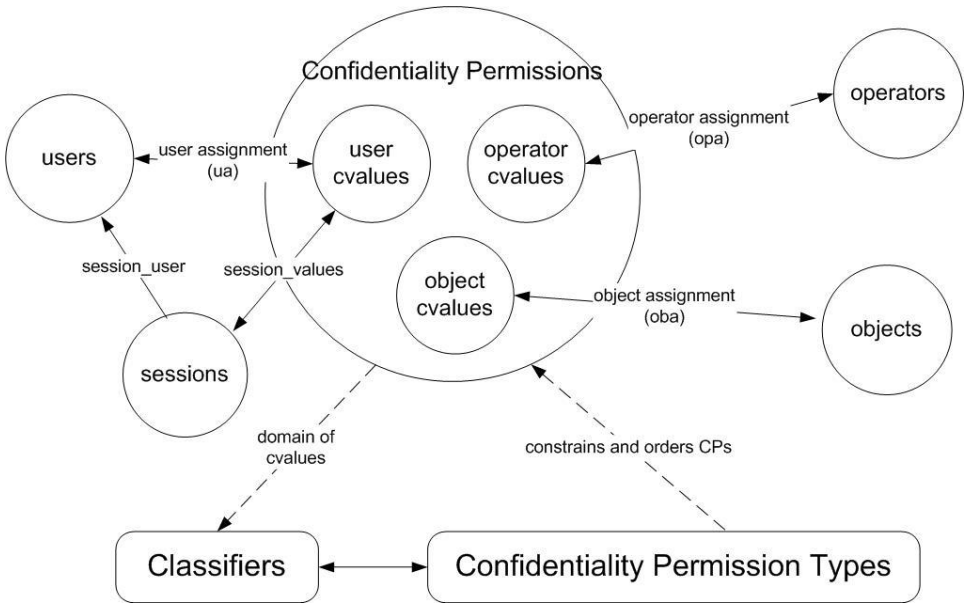


Figure 6: Core TCM

Figure 6 shows the basic elements of Core TCM. The double-headed solid arrow represents a relation (or a many-to-many relationship). The single headed solid arrow represents a function (or a many-to-one relationship). In this, there is correspondence with the RBAC model. However, although this diagram has a basic resemblance to the diagrams for RBAC, there is an essential difference in the way permissions are given or denied.

The set of Classifiers is the domain of the set of all classifier values (this is represented by one of the dashed arrows). The classifiers are derived from the persistent classifier values made available in the system e.g. there is a role classifier because the roles GP, Engineer etc. exist and are useful for authorisation.

Users are assigned to one or more classifier values (*ua*). These could be roles, locations or any other user classifier values useful for authorisation in the particular model under consideration. Each session is associated with a single user (session_user) and is related to possibly many classifier values (session_values). The classifier values that a user uses during a session must be a subset of the user‟s assigned classifier values.

There is a many-to-many relationship between the set of classifiers and the set of confidentiality permission types. The function of CPTs (in TCM2) is to constrain the Confidentiality Permissions that can be created to ones belonging to those particular types. This is represented by a dashed arrow in the diagram.

The diagram attempts to represent the relationship between user classifier values, operator classifier values, object classifier values, and confidentiality permissions. The relationship is simple but subtle, and represents an expansion of the RBAC view of authorisation.

In RBAC, permissions are assigned to roles. This allows the permissions to follow an organisation‟s structure and simplifies administration e.g. if someone leaves then the new person has to be simply assigned to the former user‟s role.

However the TCM is able to model statements like “Joe Smith is allowed access to Alice Johnson‟s Obstetric Records at James Cook University Hospital if Joe Smith belongs to Alice‟s GP Practice and is accessing the Data from the Practice Premises” .The TCM is able to handle this (and any other authorisation statement) in a straightforward way and does not need to be confined to a simpler role-structured model.

The TCM is a generalisation of RBAC because an RBAC statement is of the form, “Joe Smith in the role of GP can access Alice Johnson‟s Obstetric records” is also a valid TCM statement. The TCM is able to model any authorisation statement by asking

- What are the classifiers in the statement?
- Are they available in the system under consideration?
- If not, how can they be made available?

The following expression gives the set of classifier values that are used for authorisation. In RBAC, it would be just the session user‟s set of roles with which he created the session. In the TCM, it is the user‟s classifier values (roles, team membership, location etc.) together with the classifer values for the requested operation and the classifier values for the object upon which the operation is to be performed.

AssignedCValues (*session*, *op*, *ob*) ≙
session_values [{*session*}] ∪ ***opa*** [{*op*}] ∪ ***oba*** [{*obj*}]

The small circles within the larger circle represent the user classifier values, operator classifier values, object classifier values included in this expression for a particular session, operation, and object.

Permission is given for the particular user in the session to perform the operation on the object through the confidentiality permissions, of which there may be many.

The larger circle represents the set of confidentiality permissions through which permission is given or denied. The reason that this diagram is not relatable to the RBAC diagram is because this occurs in a much more subtle way (although still corresponding to the real world) than in RBAC.

The expression below is the essence of the matter, although there are other connotations and variations depending on whether you are dealing with TCM2 or TCM3 and including hierarchies or not etc.

It says that, if for every classifier in the confidentiality permission *cp*, there exists at least one classifier value common to both *cp* and the assigned classifier values *acvals* then *cp* permits access. Examples are given throughout this document.

CPPermitAccess (*cp, acvals*) ≙ ***bool*** (***dom*** (*acvals* ∩ *cp*) = ***dom*** (*cp*))

where *acvals* is the *AssignedCValues* from the first expression.

Although this expression is more subtle than RBAC, if simplified, it becomes RBAC.

TCM2 was developed to have a fixed set of Confidentiality Permission Types (CPTs) following the pattern of the original TCM ideas. An ordering on the CPTs is used to process the permissions given by the different types. There is a default ordering on the CPTs that is derived from a user defined ordering on the classifiers. This default ordering is based on the complexity of the corresponding CPTs, as measured by the number of classifiers in the CPT; and where two CPTs have the same number of classifiers, the relative importance of the classifiers in the two CPTs (see 3.2.2.11).

This default ordering allows first consideration to be given to exceptional circumstances, before authorisations that are more general are invoked. Therefore, a CP granting access to

GP Fred for Alice‟s psychosis data might be processed before a CP occurring lower in the processing order, which denied access to this data for all GPs.

The TCM is defined in terms of four model components: Core TCM, Hierarchical TCM, Constrained TCM, and Extended TCM. Core TCM defines those concepts (variables, functions and relations) required to achieve authorisation through classifier values. Additionally those invariants required for consistency are specified, and the preconditions that need to be attached to operations for those invariants to hold.

Hierarchical TCM introduces inheritance of permissions according to classifier values through a parent/ child relation. A single parent/ child relation for classifier values in the TCM models inheritance structures for all user, operation, and object classifiers.

Constrained TCM introduces constraints on the classifier values, which can be used for modelling separation of duties and other requirements.

Extended TCM adds derived classifier values such as <Legitimate_Relationship, Yes> as well as other classifier values for example <Security_Code, Red>. A classifier value can be any persistent value in the system that is useful for authorisation.

3.2.2 Core TCM2

3.2.2.1 Types

The B specification of Core TCM uses six types or classes:

$$USER;\ OP;\ OB;\ CFIER;\ VALUE;\ SESSION$$

A type in B is used for type checking during verification. They would correspond to classes in dotnet. Thus, there is a user class, an operation class etc.

3.2.2.2 Classifiers

A classifier is a concept that is used to define authorisation functionality for a user, operation, or object. It corresponds to a property that is considered useful for giving or denying access either on its own or in combination with other classifiers. Examples of user classifiers would be Role, Team, Location, and Work_Area. Examples of object classifiers could be data type, data location.

3.2.2.3 Classifier Values (CValues)

The type for classifier value is defined as the cross product of CFIER and VALUE

$$CVALUE \triangleq CFIER \times VALUE$$

Examples of user classifier values would be <Team, Surgical>, <Role, Administration>, <Role, GP>, <Location, Tees Valley Health Authority>, and <Location, James Cook University Hospital>.

3.2.2.4 Users, Operations, Objects

The variable *users* refers to people, machines, networks, or intelligent autonomous agents.

$$users \subseteq USER$$

i.e. *users* is a collection (set) of type *USER,* or in dotnet terms *users* is a set of instances of the class *USER.* The set of *users* are assigned to classifier values (cvalues) using the relation *ua*.

$$ua \in users \leftrightarrow CVALUE$$

Examples of user assignment would be the triples <Fred, Role, Administration>, <Fred, Team, Surgical>, <Bob, Location, James Cook University Hospital >. Operations and objects are similarly assigned to classifier values, where the classifiers are considered useful for authorisation.

$$opa \in OP \leftrightarrow CVALUE$$
$$oba \in OB \leftrightarrow CVALUE$$

An example of object assignment would be the triple <EHR1, PatientId, YS440953C>. This states that the object EHR1 (where EHR stands for Electronic Health Record) is assigned to the CValue <PatientId, YS440953C>, where YS440953C is a National Insurance Number. Of course, in a database this is just saying there is a many-to-one relationship between Electronic Health Records and Patients. However, it is useful to look at in this way, because we go on to gather all of the classifier values together, from whatever source, and it is those classifier values that are used for authorisation.

3.2.2.5 Sets of Classifier and Classifier Values

The expression ***ran****(ua)* gives the set of all user classifer values e.g. <Team, Surgical>, <Role, Administration>, <Location, Newlands Medical Practice> etc. Similarly ***ran*** *(opa)* gives the set of all operation classifer values and ***ran*** *(oba)* gives the set of all object classifer values. Therefore, the set of all classifier values (*cvalues*) is given by:

$$cvalues \triangleq \mathbf{ran}\ (ua) \cup \mathbf{ran}\ (opa) \cup \mathbf{ran}\ (oba)$$

The expression **dom** *(***ran** *(ua))* extracts the set of user classifiers from the set of user classifier values. Similarly, the operation and object classifiers are given by **dom** *(***ran** *(opa))* and **dom** *(***ran** *(oba))*.

$$user\_cfiers \triangleq \textbf{\textit{dom}}\ (\textbf{\textit{ran}}\ (ua))$$

$$op\_cfiers \triangleq \textbf{\textit{dom}}\ (\textbf{\textit{ran}}\ (opa))$$

$$obj\_cfiers \triangleq \textbf{\textit{dom}}\ (\textbf{\textit{ran}}\ (oba))$$

The set of all classifiers is given by:

$$cfiers \triangleq user\_cfiers \cup op\_cfiers \cup obj\_cfiers$$

3.2.2.6 Confidentiality Permission

A Confidentiality Permission (CP) is a set of classifier values by which authorisation is permitted or denied. An example could be:

CP = {<Team, Surgical>, <Role, Anaesthetist>, <EHR_objs, Jane>, <EHR_obj_type, Obstetrics>}

where Team and Role are user classifiers, EHR_objs and EHR_obj_type are object classifiers. The CP is assigned to permit or deny access. If the example given above were created to permit access, then it would allow the anaesthetist member of the surgical team access to patient Jane‟s obstetric data.

$\mathbb{F}(S)$ is the set of all finite subsets of S. As each CP is a subset of the set of classifier values *cvalues*, then the set of all Confidentiality Permissions *cps* is given as follows:

$$cps \subseteq \mathbb{F}\ (cvalues)$$

As stated above CPs can either permit or prohibit authorisation. There is a function from the set of CPs to the set {permit, deny} which establishes the authorisation value of each CP. Because it is a function, a CP cannot simultaneously permit and deny authorisation.

$$acps \in cps \rightarrow \{permit, deny\}$$

It is sometimes easier to work with permit and deny CP sets (or views in database terminology). The sets are defined as follows using the range restriction operator ($\rhd$):

$$permit\_cps \triangleq \textbf{\textit{dom}}\ (acps \rhd \{\text{permit}\})$$

$$deny\_cps \triangleq \textbf{\textit{dom}}\ (acps \rhd \{\text{deny}\})$$

3.2.2.7 Sessions

Every user can have a number of sessions. A partial function is defined from *SESSION* to *users*:

$$session\_user \in SESSION \nrightarrow users$$

The set of *sessions* is defined as the domain of this function:

$$sessions \triangleq \textbf{\textit{dom}}\ (session\_user)$$

and *userSessions* as its inverse:

$$userSessions\ (user) \triangleq session\_user^{-1}[\{user\}]$$

Each session has an associated set of user classifier values that must be a subset of the user"s assigned classifier values:

$$session\_values \in SESSION \leftrightarrow cvalues$$
$$session\_values \subseteq (session\_user\ ;\ ua)$$

This follows the RBAC specification with classifier values being used instead of roles.

3.2.2.8 Permitting or Denying Permission

The assigned classifier values for a particular session, operation, and object are given as follows:

$$\textbf{\textit{AssignedCValues}}\ (session, op, ob) \triangleq$$
$$\textbf{\textit{session\_values}}\ [\{session\}] \cup \textbf{\textit{opa}}\ [\{op\}] \cup \textbf{\textit{oba}}\ [\{ob\}]$$

These classifier values together with the CPs determine whether the operation *op* can be performed on the object *ob* in the given *session*. A particular permit_cp *cp* permits access according to the following expression:

$$\textbf{\textit{CPPermitAccess}}\ (cp, acvals) \triangleq \textbf{bool}\ (\textbf{dom}\ (acvals \cap cp) = \textbf{dom}\ (cp))$$

where *acvals* is the assigned classifier values (*AssignedCValues* (*session*, *op*, *ob*) as defined above), and *cp* is a confidentiality permission which grants access. Access is permitted if for every classifier in the CP, there is at least one classifier value common to the CP and the Assigned Classifier Values.

The domain of any CP is the set of classifiers used by that CP. For example the CP

{<Role, GP>, <Location, James Cook Hospital>, <EHR_obj_type, Obstetrics>}

uses the classifiers

{Role, Location, EHR_obj_type}.

The expression for *CPPermitAccess* above states that access is permitted if for every classifier in the CP there is at least one classifier value common to the CP and the assigned classifier values. As an example, suppose that:

session_values ={<Role, GP>, <Role, Administrator>, <Location, James Cook Hospital>}
op_value = {<EHR_ops, EHR_Update>}
obj_values = {<EHR_objs, Jane>, <EHR_obj_type, Obstetrics>}

then either of the two following permit_cps would permit the given operation on the given object because for every classifier in the CP there is a match between the session, operator and object classifier values, and the Confidentiality Permission classifier values.

PCP1={<Role, Administrator>, <EHR_ops, EHR_Update>, <EHR_obj_type, Obstetrics>}
PCP2={<Role, GP>, <EHR_ops, EHR_Update>, <EHR_objs, Jane>}

A CP would normally contain at least one user, operator, and object classifier value and in early versions of the TCM, this was a requirement. As an aside, it is worth noting here that the operator and object classifiers could be simply operator and object identifiers, and these together with Role as a user classifier would correspond to RBAC.

During the formal specification of the TCM, it was realised that a CP could be any set of classifier values and did not have to be restricted to having at least one user, operation, and object classifier value. This flexibility allows a CP to be specified as:

PCP3 = {<Role, GP>, <Location, James Cook Hospital>, <EHR_objs, Jane>}

or even just

PCP4 = {<Role, Administrator>}

PCP3 would allow any GP at James Cook Hospital to perform any operation on Jane‟s data. PCP4 would allow the administrator to perform any operation on any object. Note that if the above were deny_cps then the operation on the object would be denied.

3.2.2.9 Confidentiality Permission Type

With each Confidentiality Permission there is associated a set of classifiers e.g. {Role, Location, EHR_obj_type}. A Confidentiality Permission Type (CPT) is defined as this set of classifiers. Any set of classifiers can be a CPT. In the TCM as first developed and in TCM2 as specified here there is a fixed set of CPTs to which the CPs had to belong. So

the CPTs acted as a constraint on which CPs could be created. However in keeping with the generalisation theme of the formal specification, in further development we can say that any set of classifier values can be a CP, and therefore any set of classifiers can be a CPT. (This is explored more fully in TCM3 (Section 4)). Any particular CPT permits access if there exists a CP belonging to that CPT which permits access.

CPTPermitAccess (*cpt, acvals*) ≙
bool (∃ *cp.* (*cp* ∈ *permit_cps* ∧ ***dom*** *(cp)* = *cpt* ∧ ***dom*** (*acvals* ∩ *cp*) = *cpt*))

i.e. if there exists at least one CP *cp* which is of type *cpt* (this is given by *dom (cp)* = *cpt*) and for every classifier in *cpt*, there is at least one classifier value common to the CP *cp* and the Assigned Classifier Values *acvals*. Similarly, for CPTDeniesAccess except that *cp* has to be a *deny_cp*.

CPTDenyAccess (*cpt, acvals*) ≙
bool (∃ *cp.* (*cp* ∈ *deny_cps* ∧ ***dom*** *(cp)* = *cpt* ∧ ***dom*** (*acvals* ∩ *cp*) = *cpt*))

The CPT {Role, EHR_ops, EHR_objs} is equivalent to authorisation by role, because the corresponding CPs take the form (for example):

{<Role, GP>, <EHR_ops, update>, < EHR_objs, Alice"s Records>},

which is just the role to permission assignment *pa* in RBAC.

3.2.2.10 Confidentiality Permission Type Ordering

A TCM application in its original form (and in its development to TCM2, which is what we are considering here) will normally define several CPTs. These CPTs have an ordering which determines which are processed first. Authorisation is given or denied according to the first match of a CPT in the ordering i.e. the first CPT for which there exists a CP that either permits or denies access. This means that lower priority CPTs do not have to be considered once a match is found.

If a match succeeds and the CPT denies access then the TCM denies access and no further CPTs are considered. If a match succeeds and the CPT permits access and does not deny access then the TCM permits access and no further CPTs are considered. If there is no match then the next CPT is examined to see if it permits or denies access. If there is no match after all the CPTs have been examined then the default is that permission is denied.

3.2.2.11 CPT Default Ordering

For the default processing order in the original TCM papers, it was stated that more complex CPTs were processed before less complex. One reason was that CPs corresponding to the more complex were intuitively more restrictive than the less complex CPs; here „complexity" is measured as the number of classifiers in the CPT e.g. {Team, Role, EHR_objs} is more complex than {Role, EHR_objs}, and precedes it.

Also in the original TCM papers, there was an ordering on the classifiers that determined the CPT ordering for equally complex CPTs i.e. those with the same number of classifiers. The ordering on the classifiers is a user-defined „importance rating" e.g. the user can decide that role is more important than location or vice versa. So {Role, EHR_obj_type} would precede {Location, EHR_obj_type} if the designer had judged that authorisation by role was more important (and therefore should be processed before) authorisation by location.

This has to be stated with mathematical precision for the purposes of this exercise. Although the rules above give a default CPT ordering this ordering can be changed by the designer, and in fact, this has been done in some practical applications. The sequence of *cpts* (*cptsq*) is given as follows where **iseq** *(cpts)* is the set of injective sequences from the set *cpts.* It is a mapping of the integers 1, 2, 3, 4… to the set *cpts*.

$$\boldsymbol{cptsq} \in \textbf{iseq}\ (cpts)$$

If the default ordering applies, it is enforced as shown below.

1. The more complex CPT is processed first. The expression below states that for any two CPTs (*cpt1*, *cpt2*) in the range of *cptsq*: if the number of classifiers in *cpt1* (**card** (*cpt1*)**)** is greater than the number in *cpt2* (**card** (*cpt2*)) then the order number in the sequence of *cpt1* is less than that of *cpt2.*

   $$\forall\ (cpt1, cpt2).\ (cpt1 \in \textbf{ran}\ (cptsq) \wedge cpt2 \in \textbf{ran}\ (\boldsymbol{cptsq}) \wedge \textbf{card}\ (cpt1) > \textbf{card}\ (cpt2)$$
   $$\Rightarrow$$
   $$\boldsymbol{cptsq}^{-1}\ (cpt1) < \boldsymbol{cptsq}^{-1}\ (cpt2)$$

2. For CPTs of equal cardinality, there is an ordering on the classifiers that determines the ordering on the CPTs. *Cfiersq* is set by the designer and is a mapping of the set of integers 1,2,3,4.... to the set of classifiers.

   $$\boldsymbol{Cfiersq} \in \textbf{iseq}\ (cfiers)$$

The expression below states that given any two CPTs (*cpt1*, *cpt2*) in the range of *cptsq* where the number of classifiers in each are equal – then having removed from each CPT the classifiers they have in common – if the lowest order classifier in *cpt1* precedes the lowest order classifier in *cpt2* then cpt1 comes before cpt2 in the CPT ordering

$\forall$ (*cpt1, cpt2*). (*cpt1* $\in$ **ran** (***cptsq***) $\wedge$ *cpt2* $\in$ **ran** (***cptsq***) $\wedge$ **card** (*cpt1*) = **card** (*cpt2*)
$\wedge$
min (***cfiertsq*** $^{-1}$[*cpt1* - *cpt1* $\cap$ *cpt2*]) < **min** (***cfiertsq*** $^{-1}$ [*cpt2* - *cpt1* $\cap$ *cpt2*])
$\Rightarrow$
cptsq $^{-1}$(*cpt1*) < ***cptsq*** $^{-1}$(*cpt2*)

For example, if Role is considered higher in precedence than Team, and Team is considered higher in precedence than Location. Given *cpt1* = {Role, Team, EHR_obj_type} and cpt2 = {Role, Location, EHR_obj_type} then removing the classifiers in common gives {Team} for *cpt1* and {Location} for *cpt2*. Because Team is higher in precedence than Location then *cpt1* comes before *cpt2* in the CPT ordering.

3.2.2.12 First Matched CPT

As discussed above (3.2.2.10) the TCM looks for the first matched CPT in the CPT ordering. This is given by:

FirstMatchCpt (*acvals*) $\triangleq$
cptsq (***min*** ({*nn* | *nn* $\in$ ***dom*** (***cptsq***) $\wedge$
$\exists$*cp*. (*cp* $\in$ *cps* $\wedge$ ***dom*** (*acvals* $\cap$ *cp*) = ***cptsq*** (*nn*))}))

where

cps $\triangleq$ *permit_cps* $\cup$ *deny_cps*

i.e. the first CPT in the sequence *cptsq* for which there exists at least one CP that either permits or denies access.

3.2.2.13 TCM Permit Access

The TCM permits access if and only if the first match CPT permits access and does not also deny access. This assumes that the system default is that "deny overrules permit" which is the normal practice in authorisation systems. However, the default can be easily changed in the TCM if required.

$$\textbf{\textit{TCMPermitAccess}}\ (acvals) \triangleq \textbf{\textit{CPTPermitAccess}}\ (\textbf{\textit{FirstMatchCpt}}\ (acvals), acvals) \wedge$$
$$\neg\ \textbf{\textit{CPTDenyAccess}}\ (\textbf{\textit{FirstMatchCpt}}\ (acvals), acvals)$$

3.2.3 Hierarchical TCM2

Hierarchies are introduced into the TCM in terms of a collection structure or inheritance relationship. e.g. Teams or Roles. The TCM uses a parent/child relationship on classifier values to model this:

$$pc \in cvalues \leftrightarrow cvalues$$

This is similar to the hierarchical relationship in RBAC except this relationship applies to any set of classifier values, with the added requirement that any mapping in *pc* would belong to the same classifier. Thus, the single variable *pc* can model a role hierarchy, a team hierarchy, and a data type hierarchy etc. The variable *pc* holds hierarchies for every classifier in the TCM application we are considering.

Examples could be

<Operating Theatre A Team, Surgical Team 1>

for a *Team* user classifier, or

< Health Care Professional, Registrar>

< Registrar, Consultant>

for a *Role* user classifier, or

< Tees Valley Health Authority, James Cook University Hospital>

for a *Location* user classifier, or

< Medical, Psychiatric>

for a *Data Type* classifier.

Inheritance is a valuable part of RBAC, allowing appropriate permissions assigned at the correct level to be distributed throughout the roles of an organisation. It is much more valuable that through a single inheritance relation permissions can be distributed through any number of classifiers, according to this simple model.

The reflective transitive closure of the parent/ child relationship defines an ancestor/ descendant relationship as in RBAC:

$$ad \triangleq pc*$$

The descendant/ ascendant relationship *da* is its inverse

$$da = pc^{*-1}$$

3.2.3.1 Confidentiality Permission Access with Inheritance

The definition for *CPPermitAccess* is exactly the same as in (3.2.2.8) except that all CP classifier values and all inherited classifier values (given by ***ad*** [*cp*]) are used to test for access, not just the classifier values of *cp*. The set ***ad*** [*cp*]) contains the original ancestor classifier values as well as the set of all descendant classifier values.

$$\textbf{\textit{CPPermitAccess}}\ (cp, acvals) \triangleq \textbf{\textit{bool}}\ (\textbf{\textit{dom}}\ (acvals \cap \textbf{\textit{ad}}\ [cp]) = \textbf{\textit{dom}}\ (cp))$$

That is access is granted if for every classifier in the domain of *cp* there exists at least one classifier value in common between the assigned classifier values *acvals* and the classifier values of *cp* and all their descendants. Similarly for *CPDeniesAccess.*

3.2.3.2 Confidentiality Permission Type Access with Inheritance

The definition for *CPTPermitAccess* is exactly the same as in (3.2.2.9) except that as above all CP classifier values and all inherited classifier values (given by ***ad*** [*cp*]) are used to test for access, not just the classifier values of *cp*.

$$\textbf{\textit{CPTPermitAccess}}\ (cpt, acvals) \triangleq$$
$$\textbf{\textit{boo l}}\ (\exists cp\ .(cp \in permit\_cps \wedge \textbf{\textit{dom}}(cp) = cpt \wedge \textbf{\textit{dom}}\ (acvals \cap ad[cp]) = cpt))$$

Similarly for CPTDenyAccess:

$$\textbf{\textit{CPTDenyAccess}}\ (cpt, acvals) \triangleq$$
$$\textbf{\textit{bool}}\ (\exists cp.(cp \in deny\_cps \wedge \textbf{\textit{dom}}(cp) = cpt \wedge \textbf{\textit{dom}}\ (acvals \cap ad[cp]) = cpt))$$

3.2.3.3 Confidentiality Permission Type Refinement

In TCM2, there are two ways in which permissions and prohibitions interact with each other. The first we have already considered – that there is an ordering on the CPTs and access is given according to whether the first match for a CPT in the CPT ordering permits or denies access. However, in the presence of a hierarchy, we also consider the interaction of permit_cps and deny_cps within a single CPT and this determines whether the CPT itself permits or denies access. This could be considered unnecessarily overcomplicated, and it was partly to address this that TCM3 was developed. However, in fairness it must be said that usually, in practice, there are only a small number of CPTs and the complexity does not cause any great practical problems.

In Core TCM2, a CPT permits access if there exists a CP belonging to that CPT which permits access. Similarly, for deny access. In this treatment a CPT can both permit and deny access and this is resolved when deciding whether the TCM as a whole permits access.

In the presence of hierarchies, the authorising CP is selected according to the nearest match within classifier value hierarchies. As an example of this, deny_cps can act to modify or refine permit_cps i.e. to allow general overall permissions to be applied, but then to remove some detailed permissions using deny_cps. We could have the following permit_cp:

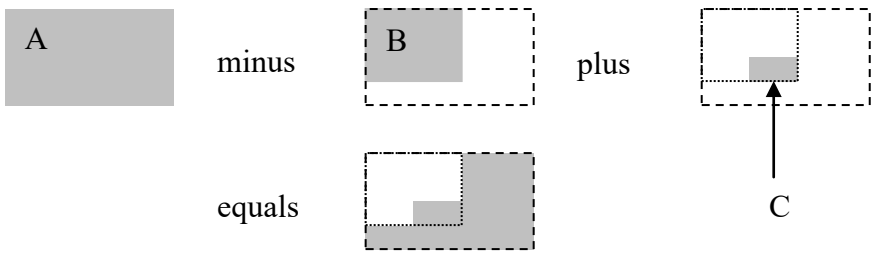
PCP={<Role,HCP>,<Location, Tees Valley Health Authority>,<EHR_obj_type, Admin>}

together with the following inheritances in *pc*:

< HCP, Locum> for Role
<Tees Valley Health Authority, James Cook Hospital> for Location
<Admin, Finance> for EHR_obj_type

The deny_cp:

DCP = {<Role, Locum>, <Location, James Cook Hospital>, <EHR_obj_type, Finance>} could be specified to remove the permission given by inheritance. This process of refinement is further illustrated in Figure 7 below.



Permit CPs

A. {<Role, HCP>, <Location, JCUH>, <EHR_type, Orthopaedic>}

C. {<Role, TraineeNurse>, <Location, JCUH_Training>, <EHR_type, Orthopaedic>}

Deny CPs

B. {<Role, TraineeNurse>, <Location, JCUH>, <EHR_type, Orthopaedic>}

Figure 7: CPT Refinement Example

Figure 7 above shows how complicated permissions can be built using simple permit and deny CPs conforming to the same CPT. This represents a situation where a trainee nurse is denied access to some data normally available to a Health Care Professional (HCP) in James Cook Hospital because of their trainee status, but is allowed access in the James Cook Hospital Training Block. Discrete points in the shaded areas represent individual authorisations. The expression for *CPTPermitAccess* becomes, for Hierarchical TCM:

CPTPermitAccess (*cpt, acvals*) ≙
bool (∃ *cp*1. (*cp*1 ∈ *permit_cps* ∧ ***dom*** (*cp1*) = *cpt* ∧ ***dom*** (*acvals* ∩ ***ad*** [*cp*1]) = *cpt* ∧
¬ (∃ *cp*2. (*cp*2 ∈ *deny_cps* ∧ ***dom*** (*cp2*) = *cpt* ∧ ***dom*** (*acvals* ∩ ***ad*** [*cp*2]) = *cpt* ∧
ad [*cp2*] ⊂ ***ad*** [*cp1*]))))

I.e. access is given if there exists a CP *cp1* belonging to *cpt* which permits access, and there does not exist a CP *cp2* belonging to *cpt* which denies access, and where the descendants of *cp2* are a subset of the descendants of *cp1*. The requirement that the descendants of *cp2* are a subset of the descendants of *cp1* means that for every classifier in the cpt the classifier value for *cp2* is a descendant of the classifier value for *cp1*. If *cp2* existed, we would say that *cp2* refines *cp1*.

If there is any CP at all that permits or denies access then there will exist a (nearest match) CP which is not refined by another CP. This is what has been meant by „nearest match" in previous TCM papers, although this was discussed in the context of an SQL implementation and described as „moving upwards until a match was found". In „limited hierarchies" a unique first match would be found. However, in general hierarchies there may be a number of first matches. This causes no difficulties providing a CPT is allowed to both permit and deny access (or a default „deny overrules permit" rule is applied).

CPTDenyAccess (*cpt, acvals*) ≙
bool (∃*cp*1. (*cp*1 ∈ *deny_cps* ∧ ***dom*** (*cp1*) = *cpt* ∧ ***dom*** (*acvals* ∩ ***ad*** [*cp*1]) = *cpt* ∧
¬ (∃*cp*2. (*cp*2 ∈ *permit_cps* ∧ ***dom*** (*cp2*) = *cpt* ∧ ***dom*** (*acvals* ∩ ***ad*** [*cp*2]) = *cpt* ∧
ad [*cp2*] ⊂ ***ad*** [*cp1*]))))

In accordance with Core TCM a CPT can both permit and deny access and this is eventually resolved when deciding whether the TCM as a whole permits access.

3.2.3.4 TCM2 Permit Access

Having redefined *CPTPermitAccess*, CPT Ordering and *TCMPermitAccess* are processed exactly as in Core TCM (3.2.2.12).

3.2.4 Constrained TCM2

In the RBAC standard (Ferraiola, et al., 2001) static and dynamic separation of duties are defined. Static separation determines those duties to which a user cannot be simultaneously assigned e.g. bank manager and bank teller. Dynamic separation specifies those duties that a user cannot hold simultaneously within a session, even though they may be assigned those duties. By duties is meant roles, and although the RBAC standard specifies a particular way of implementing separation of duties, essentially there are sets of roles which together are prohibited, either on assignment or within a session. The TCM specification varies from the RBAC approach to reflect this.

Otherwise the RBAC approach is extended quite naturally in the TCM to static and dynamic separation of classifier values (as opposed to roles), and two variables are implemented which contain the banned sets of classifier values. Although I use *ssv* and *dsv* as in RBAC, the terms stand for different things i.e. banned sets of classifier values, rather then a partial function between classifier values (in RBAC roles) and a natural number *nn*.

$$ssv \subseteq \mathbf{F}(cvalues)$$
$$dsv \subseteq \mathbf{F}(cvalues)$$

As in RBAC, static separation requires an additional precondition to *AssignUser* and dynamic separation requires an additional precondition to both *CreateSession* and *ActivateValue*.

3.2.5 Extended TCM2

Extended TCM2 includes extra classifier values in addition to those directly related to user, operator, or object. For example <Legitimate_relationship, yes> is derived from the session and the object. System classifier values such as <Security_code, red> can also be included. The relation *ea* (extended assignment) is defined below. Although included here as part of TCM2 these ideas are developed more fully in TCM3 (Section 4), and it was some of this thinking about TCM2 that partly led to TCM3‟s development.

$ea \in SESSION \times OP \times OB \leftrightarrow CVALUE$

The extended classifiers are given by:

$ext\_cfiers \triangleq \mathbf{dom}\ (\mathbf{ran}\ (ea))$

and the set of all classifiers becomes

$cfiers \triangleq user\_cfiers \cup op\_cfiers \cup obj\_cfiers \cup ext\_cfiers$

The extended classifier values are added to the assigned classifier values for session, operator and object and are used for authorisation together with the session, operator and object classifier values

$\mathbf{AssignedCValues}\ (session, op, ob) \triangleq$
$\mathbf{session\_values}\ [\{session\}] \cup \mathbf{opa}\ [\{op\}] \cup \mathbf{oba}\ [\{obj\}] \cup \mathbf{ea}\ [\{session \mapsto op \mapsto ob\}]$

3.2.6 TCM2 Overrides

The TCM2 user can apply an override in two basic ways and in a combination of the two. The overrides act to remove denial effects.

- ***CP Override.*** The removal of the denial effect of a deny CP upon a permit CP within a CPT. An example could use teams and sub teams. e.g. we could have the permit_cp

PCP1 = {<Team, T1>, <EHR_obj_type, Surgical>}

with the following team inheritance relations in *pc*:

<T1, T2>
<T2, T3>

If there exists a deny_cp

DCP1 = {<team, T3>, <EHR_obj_type, Surgical>}

then this cancels the collection to sub-collection inheritance at level T3. The override would allow any users classified as belonging to T3 to inherit from the permit_cp. CP Override can be used with any user classifier or combination of user classifiers. It is equivalent to substituting the session value in *acvals* with the level to which the CP Override takes place, in this example replacing T3 with T2. Additional examples could be given for Role (e.g. substituting GP for Junior GP), or other session values.

- ***CPT Override.*** This has the effect of cancelling any denials produced by CPs corresponding to earlier CPTs to a specified „override level" in the CPT processing order.

The first match would only look at permit CPs in the given set and then permit or deny CPs thereafter. CPT Override can be applied in a number of ways depending on the application. It might remove denials by specialist permissions, leaving the user to access information granted by a more general permission providing role-based access. The first matched CPT for a set of assigned classifier values now becomes:

FirstMatchCpt (*acvals, cpt_override_level*) ≜
cptsq (***min*** ({*nn* | *nn* ∈ ***dom*** (***cptsq***) ∧
(*nn* <= *cpt_override_level* ∧ ∃*cp*. (*cp* ∈ *permit_cps* ∧ ***dom*** (*acvals* ∩ *cp*) = ***cptsq*** (*nn*))}
∨
(*nn* > *cpt_override_level* ∧ ∃*cp*. (*cp* ∈ *cps* ∧ ***dom*** (*acvals* ∩ *cp*) = ***cptsq*** (*nn*)))))}))

where

cps ≜ *permit_cps* ∪ *deny_cps*

- ***Global Override.*** The removal of all denial effects, so that the expression for *TCMPermitAccess* simplifies to:

TCMPermitAccess (*acvals*) ≜
bool (∃*cp*. (*cp* ∈ *permit_cps* ∧ ***dom*** *(cp)* = ***dom*** (*acvals* ∩ *cp*)))

3.2.7 Positive/ Negative Permissions Conflict Solving

The procedure for handling conflicts between positive and negative permissions is restated here, although it is contained in the descriptions and mathematics below. This is because this topic has been the subject of a number of papers, and because TCM3 handles this differently to TCM2 so the differences need to be highlighted.

In TCM2, there are two areas where positive and negative permissions interact:

- Within a Confidentiality Permission Type
- Between Confidentiality Permission Types

3.2.7.1 Within a Confidentiality Permission Type

A particular CPT denies access if and only if there exists a CP belonging to that CPT which denies access and that CP is not refined by a CP that permits access. Refines is given by 3.2.3.3

A particular CPT permits access if and only if there exists a CP belonging to that CPT which permits access and that CP is not refined by a CP that denies access.

In the light of the above two statements a CPT can permit access, deny access, both permit or deny access, or neither permit or deny access. (In deference to some comments, you could apply the default that „deny overrules permit" or vice versa within a CPT, so then a CPT can be said to only permit or deny access. However, what matters is whether the TCM overall permits or denies, and this logic would still achieve the same outcome but by a slightly different route.)

3.2.7.2 Between Confidentiality Permission Types

There is an ordering on the CPTs that determines whether the TCM as a whole grants or denies access. This is known as the first match principle. Each CPT is examined in order of priority. The following rules apply:

- If the first match CPT denies access then access is denied by the TCM, irrespective of whether the CPT also permits access
- If the first match CPT permits access and does not deny access then access is permitted by the TCM
- If the CPT neither permits access nor denies access then the next CPT in the ordering is examined.
- If all the CPTs are examined and no CPT is found which permits access or denies access then the TCM denies access.

3.3 TCM2 Functional Specification in B: Overview

3.3.1 Introduction

This section defines features required of a TCM system. These features fall into three categories: administrative operations, administrative reviews, and system-level functionality.

The administrative operations define requirements in terms of an administrative interface and an associated set of semantics that provide the capability to create, delete, and maintain TCM elements and relations (e.g., to create and delete classifier value to CP assignments).

The administrative review features define requirements in terms of an administrative interface and an associated set of semantics that provide the capability to perform query

operations on TCM elements and relations, for example, return the set of users assigned to a classifier value or the set of classifier values that are either assigned to or inherited by a user.

System-level functionality defines features for the creation of user sessions to include classifier value activation/ deactivation, the enforcement of constraints on classifier activation, and for calculation of an access decision.

3.3.2 Core TCM2 Operations

3.3.2.1 Administrative Operations for Core TCM

Administrators create and delete users, assign and deassign users to classifier values (which could include roles), and add or remove Confidentiality Permissions. Confidentiality Permissions give authorisation through user classifier values in combination with operation and object classifier values.

— AddUser: adds a user to the set *users*. Details of users are usually inputted before any authorisation is assigned, and may be kept on the system after they have left the company.
— DeleteUser: deletes a user.
— AssignUser: assigns a user to a classifier value. This could be a role, location, team, or any attribute the user could have.
— DeassignUser: deassigns a user from a classifier value.
— AddCP: adds a Confidentiality Permission.
— RemoveCP: removes a Confidentiality Permission.

3.3.2.2 Supporting System Operations for Core TCM

— CreateSession: creates a user session and gives the user a set of classifier values, which must be a subset of the user"s assigned classifier values.
— DeleteSession: terminates a session.
— ActivateValue: adds a classifier value as an active classifier value for the current session.
— DeactivateValue: deletes a classifier value from the active value set of the current session.
— CPPermitAccess: returns TRUE if the given CP permits access.
— CPDenyAccess: returns TRUE if the given CP denies access.
— CPTPermitAccess: returns TRUE if the given CPT permits access.

— CPTDenyAccess: returns TRUE if the given CPT denies access.

— TCMPermitAccess: returns TRUE if the Tees Confidentiality Model as a whole permits access.

3.3.2.3 Review Operations for Core TCM2

— AssignedUsers: returns the set of users assigned to a given classifier value.

— UserCVValues: returns the set of classifier values assigned to a given user.

— AssignedCVValues: returns the set of classifier values assigned for given session, operation, and object. These classifier values together with the CPs determine whether the operation is allowed on the object in the session.

— SessionPermissions: returns the set of permissions available in the given session:

— SessionOperationsOnObject: returns the set of operations that can be performed on the given object in the given session:

3.3.3 Hierarchical TCM2 Operations

The operations *CPPermitAccess* and *CPDenyAccess* are modified to enable authorisation by a CP and all inherited values i.e. using ***ad*** [*cp*] instead of *cp*. Additionally there is a „nearest match" rule operating i.e. the permission is determined by the CP in the nearest order in the classifier value hierarchies. *CPPermitAccess* is modified to enable this. Where *CPPermitAccess* and *CPDenyAccess* are included in other operations, these are also modified. All other operations remain the same.

3.3.3.1 Additional Administrative Operations for Hierarchical TCM2

The additional administrative operations for Hierarchical TCM are concerned with the creation and maintenance of the parent child relationship *pc*.

— AddInheritance: establishes a new immediate inheritance relationship between two classifier values. The classifier values must both belong to the same classifier.

— DeleteInheritance: deletes an existing immediate inheritance relationship between two classifier values.

3.3.4 Constrained TCM2 Operations

The operation *AssignUser* is modified to ensure that the set of assigned user values does not contain a set of statically prohibited classifier values after the operation. The operations *CreateSession* and *ActivateValue* are modified to ensure that the set of session

values does not contain a set of dynamically prohibited classifier values after those operations. All other operations remain the same.

3.3.4.1 Additional Administrative Operations for Constrained TCM2

The additional operations are to do with the creation and deletion of prohibited sets of classifier values, either static i.e. prohibited for user assignment, or dynamic i.e. prohibited for use within a session.

- AddSSVSet: adds a set of statically prohibited classifier values.
- RemoveSSVSet: deletes a set of statically prohibited classifier values.
- AddDSVSet: adds a set of dynamically prohibited classifier values.
- RemoveDSVSet: deletes a set of dynamically prohibited classifier values.

3.4 TCM2 Functional Specification in B

Many of these operations in the TCM parallel the RBAC specification e.g. instead of dealing with session roles for a user, we are dealing with session values. The main differences occur in the operations that grant or deny access.

3.4.1 Core TCM2

3.3.4.1 Administrative Operations for Core TCM2

AddUser

This operation creates a new TCM user. The operation is valid only if the new user is of type USER and not already a member of users. The new user is created before classifier values are assigned or sessions are created for that user.

AddUser (user) ≙
  PRE
    user ∈ USER - users
  THEN
    user ≔ users ∪ {user}
  END

DeleteUser

This operation deletes a user from the TCM. The user assignment relation (*ua*) and the *session_user* function are updated. The user‟s sessions are removed from the *session_values* relation.

DeleteUser (user) ≙
  PRE

user ∈ users
THEN
users ≔ users - {user} ||
ua ≔ {user} ⩤ ua ||
session_user ≔ session_user ⩥ {user} ||
session_values ≔ userSessions (user) ⩤ session_values
END

AssignUser

This operation assigns a classifier value to a user. The operation is valid if the user is a member of users and the assignment does not already exist.

AssignUser (user, cfier, val) ≙
PRE
user ∈ users ∧ cfier ∈ CFIER ∧ val ∈ VALUE ∧
user ↦ (cfier ↦ val) ∉ ua
THEN
ua ≔ ua ∪ {user ↦ (cfier ↦ val)}
END

DeassignUser

This operation deletes the assignment of a classifier value to a user. The classifier value must be one of the user"s assigned classifier values and not currently active in a session.

DeassignUser (user, cval) ≙
PRE
user ∈ users ∧
cval ∈ UAssignedcvalues [2](user) – CValuesActive[3] (user)
THEN
ua ≔ ua – {user ↦ cval}
END

AddCP

This operation adds a Confidentiality Permission that acts to permit or deny access. A CP consists of a set of classifier values i.e. is in **F** (CVALUE) which is the set of all finite subsets of CVALUE and must not already be in the set *cps*. There is a function *acps* from each CP to BOOL according to whether the CP permits or denies access.

AddCP (cvals, PorD) ≙

[2] UAssignedcvalues(u) == ua[{u}]
[3] CValuesActive(u) == session_values[session_user~[{u}]]

PRE
cvals ∈ 𝔽(CVALUE) - cps ∧
PorD ∈ {permit, deny}
THEN
cps ≔ cps ∪ {cvals} ||
acps (cvals) ≔ PorD
END

RemoveCP

This operation removes a CP, which by definition is a set of classifier values. The CP must already be a member of *cps*; and the *acps* function, which assigns permit or deny to the CP, is updated using the domain anti-restriction operator.

RemoveCP (cvals) ≙
PRE
cvals ∈ cps
THEN
cps ≔ cps - {cvals} ||
acps ≔ {cvals} ⩤ acps
END

3.4.1.2 Supporting System Operations for Core TCM

CreateSession

This operation creates a new session for the given *user*, with the given set of user classifier values. (This is similar to RBAC except that RBAC uses a set of roles.) The operation is valid if and only if the *user* is a member of the *users* set, the *session* is a new session, the active value set (*avs*) is a subset of the user"s assigned classifier values, and there is at least one classifier value for each user classifier. In RBAC, this last condition is always met because there is only one classifier „role" and there is always a value for it. Here it must be enforced to enable the permissions to be correctly calculated. In a database implementation this could be done by having default values for all the user classifiers

CreateSession (user, session, avs) ≙
PRE
user ∈ users ∧
session ∈ SESSION – Sessions[4] ∧
avs ⊆ UAssignedcvalues (user) ∧
dom (avs) = user_cfiers

[4] Sessions == **dom** (session_user)

THEN

session_user (session) ≔ user ||

session_values ≔ session_values ∪ {session} x avs

END

DeleteSession

This operation deletes a *session*. It is valid if the *session* belongs to the set *Sessions*, which is defined as the domain of *session_user*. The relations *session_user* and *session_values* are modified using domain subtraction

DeleteSession (session) ≙

PRE

session ∈ Sessions

THEN

session_user ≔ {session} ⩤ session_user ||

session_values ≔ {session} ⩤ session_values

END

ActivateValue

This operation activates a given classifier value for a session. The session must already exist and the classifier value must be one of the session user"s assigned classifier values, and not currently active in the session.

ActivateValue (session, cvalue) ≙

PRE

session ∈ Sessions ∧

cvalue ∈ Uassignedcvalues (session_user (session)) -

ActiveCvalues (session)

THEN

session_values ≔session_values ∪ {session ↦ cvalue}

END

DeactivateValue

This operation deletes a classifier value from the active classifier values of a *session*. The operation is valid if and only if the *session* is a member of *Sessions*, and the classifier value is an active classifier value of that *session*.

DeactivateValue (session, cvalue) ≙

PRE

session ∈ Sessions ∧

cvalue ∈ ActiveCvalues (session)

THEN

session_values ≔ session_values - {session ↦ cvalue}
END

CPPermitAccess

This operation returns true if the given CP *cp* permits access for the given set of assigned classifier values *acvals*. The set *acvals* is the set of user, operation, and object classifiers, which are used for permission determination. Access is given if *cp* is a *permit_cp* and for every classifier in *cp* there is at least one classifier value in common between *acvals* and the *cp* (3.2.2.8).

CPPermitAccess (cp, acvals) ≙
PRE
cp ∈ permit_cps ∧ acvals ∈ **F**(cvalues)
THEN
pa ≔ **bool** (**dom** (acvals ∩ cp) = **dom** (cp))
END

CPDenyAccess

This operation returns true if the given CP *cp* denies access for the given set of assigned classifier values *acvals*. The set *acvals* is the set of user, operation, and object classifiers that are used for permission determination. Access is given if *cp* is a *deny_cp* and for every classifier in *cp* there is at least one classifier value in common between *acvals* and the *cp* (3.2.2.8).

CPDenyAccess (cp, acvals) ≙
PRE
cp ∈ deny_cps ∧ acvals ∈ **F**(cvalues)
THEN
pa ≔ **bool** (**dom** (acvals ∩ cp) = **dom** (cp))
END

CPTPermitAccess

This operation returns true if the given CPT (*cpt*) permits access for the given set of assigned classifier values *acvals*. This is true if there is at least one CP *cp* belonging to the CPT that permits access.

CPTPermitAccess (cpt, acvals) ≙
PRE
cpt ∈ cpts ∧ acvals ∈ **F**(cvalues)
THEN
pa ≔ **bool** (∃cp. (cp ∈ permit_cps ∧ **dom** (cp) = cpt ∧

dom (acvals ∩ cp) = cpt))
END

CPTDenyAccess

This operation returns true if the given CPT (*cpt*) denies access. This is true if there is at least one CP *cp* belonging to the CPT that denies access.

CPTDenyAccess (cpt, acvals) ≙
PRE
cpt ∈ cpts ∧ acvals ∈ **F**(cvalues)
THEN
pa :=**bool** (∃cp. (cp ∈ deny_cps ∧ **dom** (cp) = cpt ∧
dom (acvals ∩ cp) = cpt))
END

FirstMatchCpt

This operation returns the first CPT in *cptsq* that either permits or denies access.

fmCpt ← **FirstMatchCpt** (acvals) ≙
PRE
acvals ∈ **F**(cvalues)
THEN
fmCpt := cptsq (min ({nn | nn ∈ **dom** (cptsq) ∧
∃cp. (cp ∈ cps ∧ **dom** (acvals ∩ cp) = cptsq (nn))}))
END

TCMPermitAccess

This operation returns true if the TCM permits access. Access is permitted if and only if it is permitted by the first CPT for which there is a match (*fmCpt*) and is not also denied by that CPT.

pa ← **TCMPermitAccess** (acvals) ≙
PRE
acvals ∈ **F**(cvalues)
THEN
pa := CPTPermitAccess(fmCpt, acvals) ∧ ¬CPTDenyAccess(fmCpt,acvals)
END

3.4.1.3 Review Operations for Core TCM2

SessionPermissions

This operation returns the permissions of a given session.

sp ← SessionPermissions (session) ≜
PRE
session ∈ Sessions
THEN
sp ≔ {op, obj | op ∈ OP ∧ obj ∈ OB ∧
TCMPermitAccess (AssignedCValues[5](session, op, obj))}
END

SessionOperationsOnObject

This operation returns the operations allowed in a given session on a given object.

so ← SessionOperationsOnObject (session, obj) ≜
PRE
session ∈ Sessions ∧ obj ∈ OB
THEN
so ≔ {op | op ∈ OP ∧
TCMPermitAccess (AssignedCValues (session, op, obj))}
END

3.4.2 Hierarchical TCM2

Hierarchies are added to the TCM through an immediate parent/ child relationship *pc*. This can apply to collections where the child is a member of the parent collection e.g. teams and team members, or to inheritance as in RBAC, e.g. GP inherits from Health Care Professional. There are new operations to add or delete inheritance, and other operations are modified so that classifier values and inherited classifier values are used for authorisation not just the classifier values themselves. All other operations remain the same.

3.4.2.1 Administrative Operations

AddInheritance

This operation establishes a new parent child relationship. The preconditions are that the relationship does not already exist and that cycle creation is avoided i.e. that the parent does not inherit from the child through *ad*.

AddInheritance (cvp, cvc) ≜
PRE
cvp ∈ cvalues ∧ cvc ∈ cvalues ∧
cvp ↦ cvc ∉ pc ∧

[5] AssignedCValues(s,p,b) == session_values[{s}] ∪ opa[{p}] ∪ oba[{b}] ∪ sa[{s}]

cvc ↦ cvp ∉ ad
THEN
pc ≔ pc ∪ {cvp ↦ cvc}
END

DeleteInheritance

This operation deletes a parent child relationship. This operation affects a user"s authorised classifier values, which in turn affects the allowed session classifier values. For simplicity I have used the precondition Sessions = ϕ. In practice the Session Permissions could be modified, could be allowed to remain until the end of the session or the session forced to terminate.

DeleteInheritance (cvp, cvc) ≙
PRE
cvp ∈ cvalues ∧cvc ∈ cvalues ∧
cvp ↦ cvc ∈ pc ∧
Sessions = φ
THEN
pc ≔ pc − {cvp ↦ cvc}
END

3.4.2.2 Supporting System Operations for Hierarchical TCM2

The ancestor/ descendant relation *ad* is defined as the closure of the parent/ child relation and is used in the following operations so that all inherited classifier values are included for authorisation.

CPPermitAccess

This operation returns true if the given CP permits access**.**

pa ← **CPPermitAccess** (cp, acvals) ≙
PRE
cp ∈ permit_cps ∧ acvals ∈ **F**(cvalues)
THEN
pa ≔ **bool** (**dom** (acvals ∩ **ad** [cp]) = **dom** (cp))
END

CPDenyAccess

This operation returns true if the given CP (including inherited values) denies access.

da ← **CPDenyAccess** (cp, acvals) ≙
PRE

cp ∈ deny_cps ∧ acvals ∈ **F**(cvalues)

THEN

da ≔ **bool** (**dom** (acvals ∩ **ad** [cp]) = **dom** (cp))

END

CPTPermitAccess

This operation returns true if the given CPT permits access. This is true if there is at least one CP associated with the CPT that permits access (with inheritance) and there is no deny CP refining the permit CP to deny access. In CP override, this denial effect would be removed.

pa ← **CPTPermitAccess** (cpt, acvals) ≙

PRE

cpt ∈ cpts ∧ acvals ∈ **F**(cvalues)

THEN

pa ≔ **bool** (∃cp1. (cp1 ∈ permit_cps ∧ **dom** (cp1) = cpt ∧

dom (acvals ∩ **ad** [cp1]) = cpt ∧

¬ (∃cp2. (cp2 ∈ deny_cps ∧ **dom** (cp2) = cpt ∧

dom (acvals ∩ **ad** [cp2]) = cpt ∧

ad [cp2] ⊂ **ad** [cp1]))))

END

CPTDenyAccess

This operation returns true if the given CPT denies access. This is true if there is at least one CP associated with the CPT which denies access (with inheritance) and there is no permit CP refining the deny CP to permit access.

da ← **CPTDenyAccess** (cpt, acvals) ≙

PRE

cpt ∈ cpts ∧ acvals ∈ **F**(cvalues)

THEN

pa ≔ **bool** (∃cp1. (cp1 ∈ deny_cps ∧ **dom** (cp1) = cpt ∧

dom (acvals ∩ **ad** [cp1]) = cpt ∧

¬ (∃cp2. (cp2 ∈ permit_cps ∧ **dom** (cp2) = cpt ∧

dom (acvals ∩ **ad** [cp2]) = cpt ∧

ad [cp2] ⊂ **ad** [cp1]))))

END

3.4.3 Constrained TCM2

Additional operations are required to add and remove sets of statically and dynamically prohibited classifier values. Additional preconditions are required for AssignUser, CreateSession, and ActivateValue to ensure that prohibited sets are not activated.

3.4.3.1 Administrative Operations

AddSSVset

This operation adds a set of classifier values to which a user is not allowed to be assigned at the same time.

AddSSVset (cvals) ≜

PRE
    cvals ∈ 𝔽(cvalues) - ssv
THEN
    ssv ≔ ssv ∪ {cvals}
END

RemoveSSVset

This operation removes a set of classifier values, to which a user is not allowed to be assigned at the same time.

RemoveSSVset (cvals) ≜

PRE
    cvals ∈ ssv
THEN
    ssv ≔ ssv - {cvals}
END

AddDSVset

This operation adds a set of classifier values, to which a user is not allowed to hold concurrently in a session even though the user may be assigned the classifier values.

AddDSVset (cvals) ≜

PRE
    cvals ∈ 𝔽(cvalues) - dsv
THEN
    dsv ≔ dsv ∪ {cvals}
END

RemoveDSVset

This operation removes a set of classifier values, which a user is not allowed to hold concurrently in a session.

RemoveDSVset (cvals) ≙

PRE

cvals ∈ dsv

THEN

dsv ≔ dsv - {cvals}

END

AssignUser

Static separation of values requires an additional precondition to this operation, which is that a banned set of static values cannot be contained in the user"s new set of assigned values.

AssignUser (user, cfier, val) ≙

PRE

user ∈ users ∧ cfier ∈ CFIER ∧ val ∈ VALUE ∧

user ↦ (cfier ↦ val) ∉ ua ∧

∀cvs. (cvs ∈ ssv ⇒ cvs ⊈ UAssignedcvalues[6](user) ∪ {cfier ↦ val})

THEN

ua ≔ ua ∪ {user ↦ (cfier ↦ val)}

END

3.4.3.2 Supporting System Operations for Constrained TCM2

CreateSession

Dynamic separation of values requires an additional precondition to this operation, which is that a prohibited set of classifier values cannot be contained in the active value set *avs*.

CreateSession (user, session, avs) ≙

PRE

user ∈ users ∧session ∈ SESSION –Sessions ∧

avs ⊆ UAssignedcvalues (user) ∧

dom (avs) = user_cfiers ∧

∀cvs. (cvs ∈ dsv ⇒ cvs ⊈ avs)

THEN

session_user (session) ≔ user ||

session_values ≔ session_values ∪ {session} x avs

END

[6] UAssignedcvalues(u) == ua[{u}]

ActivateValue

Dynamic separation of values requires an additional precondition to this operation, which is that a banned set of classifier values cannot be contained in the new active value set.

ActivateValue (session, cvalue) ≙

PRE

session ∈ Sessions ∧

cvalue ∈ UAssignedcvalues (session_user (session))

- ActiveCValues[7](session) ∧

∀cvs. (cvs ∈ dsv ⇒ cvs ∉ ActiveCValues (session) ∪ {cvalue})

THEN

session_values ≔ session_values ∪ {session ↦ cvalue}

END

3.5 Comparison with RBAC and related work

3.5.1 Introduction

Because the TCM is a generalization of RBAC, it can provide all the benefits of RBAC. A CPT with a single user classifier of Role, and with operation and object identity classifiers, directly implements the concept of role authorisation found in the NIST standard.

A large research development in RBAC can be described under the term parameterised RBAC. Part of this work involves using external (sometimes called contextual, environmental) information to control the processing of roles, and therefore provides additional functionality over standard RBAC, including what could be described as an override capability (Stermbeck, et al., 2004) (Bacon, et al., 2001).

The TCM provides aspects of external parameter handling as part of its basic model and design framework (in that classifiers can represent external parameters). This approach is similar to that advocated in (Goh, et al., 1998), which prefers the use of „role attributes" to the use of external policy-enforcing systems (where this is possible).

The modelling of users and user groups for RBAC systems was reported in (Osborn, et al., 2000). However the objective in this work is to design RBAC authorisation systems, in that user groups are associated with roles. The TCM provides the functionality to

[7] ActiveUCValues(s) == session_values[{s}]

authorise by user groups with the same hierarchical structure as used in (Osborn, et al., 2000), in conjunction with Role and other classifiers.

An approach to authorising by team and role was investigated in the TMAC model (Thomas, 1997). This relied on ultimately authorising by RBAC roles and permissions, though permission activation was constrained to individual users and objects. The TCM could be used to model and implement this approach within its confidentiality permissions processing framework.

RBAC models for role administration (i.e. for assigning roles to users) have been extensively researched, e.g. the ARBAC02 model (Oh, et al., 2006). The ARBAC02 model includes models of organisation structures for user pools and permission pools that are independent from role hierarchies. These concepts could in principle be modelled in the TCM by classifiers, which are independent. The development of a system for TCM administration is beyond the scope of this thesis, and is a topic for continuing research.

The TCM can straightforwardly support a central concept of usage control (Zhang, et al., 2005) in that mutable attributes can be modelled as classifiers, and can participate in CPTs. Legitimate Relationship is similar to a mutable attribute, and can determine and change access during a session. Additionally, Extended TCM can model the relationship between a user and a protected object, unlike UCON; the Legitimate Relationship classifier is an example of this.

Access control based on credentials, and modelling of trust has been reported in the TrustBAC system (Chakraborty, et al., 2006). Here trust levels, based on credentials and other information, are mapped onto RBAC roles. The TCM provides the additional capability to model and use these concepts directly and independently for authorisation, if such functionality were to be required.

Neumann and Strembeck (Neumann, et al., 2002) have advocated using scenarios and sequence diagrams to derive permissions, and assign permissions to functional roles (as opposed to organisational roles). Their approach has had a significant effect on healthcare computing (Science Applications International Corporation, SAIC, 2005). The TCM design procedure is similar in that we use models developed during systems analysis, but we depart from RBAC permissions and roles.

3.5.2 Separation of Duties

In relation to static separation of duties: in the RBAC standard (INCITS, 2004) this is specified by giving a set of roles together with an integer *n*, with the requirement that you can"t have *n* or more of these roles assigned at the same time. This same basic algorithm applies whether the roles are statically or dynamically assigned.

However, if you take this back to basics, all that is really happening, is that there are a number of sets of roles that are not allowed to be assigned. Of course, the disallowed sets can be specified as in the RBAC standard by using rolesets and natural numbers, but mathematically all that is happening is that there are a number of particular sets of roles that are prohibited. One review of this work (an unsuccessful ACM Tissec submission in 2007) says:

"The authors talk about separation of duties. The formalization of separation of duties doesn't seem correct. Separation of duty is specified between users, objects, and operations - this doesn't seem intuitive"

The formalisation is correct, and reflects the discussion above. Perhaps I could have explained it better in the paper. If in RBAC, separation of duties is really Sets of {Sets of Roles} that are not allowed then the corresponding idea in the TCM is Sets of {Sets of Classifier Values} that are not allowed (Section 3.4.3.1). If we confine the classifier values to just user classifier values, then I think the generalisation is easily seen: in addition to sets of roles that are not allowed, there may be (for example) certain roles that are not allowed in certain locations.

However, in keeping with the generic nature of the TCM, there is no restriction that the classifier values have to be user classifier values. Although less immediately applicable, a moment"s thought could give examples where certain roles are not allowed to interact with particular data types, or certain operations are not allowed in certain locations.

This reviewer has failed to grasp the breadth of the TCM, which is made possible by taking Access Control back to first principles. This was made possible by formal specification.

3.5.3 Attributes

Some criticism of the TCM has said that classifiers are only attributes by another name and that RBAC with attributes is well covered elsewhere and provides a more powerful model than the TCM. This does not take into account Classifiers that are not dependant simply

on a single attribute. E.g., Legitimate Relationship depends on Role and the Data. The TCM is generic and allows fundamental redesign of how authorisation is given. What happens if it is decided that Role is not a factor in Authorisation e.g. Authorisation only depends on location. Of course, you can define a role „LocationPerson" but you are already adding unnecessary complexity. It would be my contention that the TCM starts by accepting intrinsic authorisation complexity, and that thereafter authorisation application is simpler, whereas RBAC starts by simplifying authorisation and that thereafter the demands of real authorisation scenarios require increasingly complex layers.

3.6 Comparison with Original TCM (TCM1)

As a result of the formal specification, some changes (which I believe are improvements) were made to the original TCM specification. The first of these concerns the fact that collections were widely used in TCM1 e.g. a CP would be applied to a collection of identities, or a collection of objects. The following is a definition used in this early work:

"A *Collection* has *Elements*, which may be *Members* or other collections. Collections and elements are uniquely identified. Collections are inherently hierarchical in that they can contain sub-collections, which in turn can have their own sub-collections. Elements can participate in more than one collection."

However, it was realised that belonging to a collection is just another classifier: in fact, access control is all about using collections to simplify administration – so a better way of looking at the above statement is that *Members* are classified as belonging to a collection (Team, Workgroup, Data Type etc.) and that inheritance is through collections and sub-collections.

TCM1 also included inheritance within a CP. To each classifier value within a CP was attached upward or downward inheritance. This was specified at the time of the CP's creation. It was realised (after lengthy discussion) that that all upward inheritance can be rewritten as downward inheritance, and any inheritance applied within a CP, can be split into specifying the CP without inheritance and specifying the inheritance in a relation. Specifying a CP as originally envisaged is still possible as part of the GUI, but the underlying programming means that all inheritance can be handled by one relation, the parent / child relation *pc*.

TCM has a first match principle for deciding which CPT to apply. Within the CPT, the original TCM (TCM1) also applied a nearest match principle to decide which CP within that CPT took priority. This worked as follows: given an ordering on the classifiers e.g. if

Role is more important than Location, which is more important then Data Type, then starting with the inheritance tree for Role and given the role of the user, you would look up the inheritance tree until a CP was found and that CP would apply. Given two CPs with the same role value you would then additionally look up the inheritance tree for Location. And so on, until a unique CP was found.

The first comment is that this obviously limits the TCM to „limited" rather than „general" hierarchies and this is not desirable if we wish to provide a generic formal specification.

However, the approach is consistent and can be described mathematically, but the additional questions that have to be asked are "Does this correspond to the real world?" and "Does this help or hinder the system administrator?" As an example suppose we have the following CPs shown in Figure 8:

A permit cp:

PCP1 = {<Role, Consultant>, <DataLocation, TVHA[8]>, <DataType, Admin>}

A deny cp:

DCP1 = {<Role, HCP>, <DataLocation, JCUH>, <DataType, Finance>} together with the following inheritance relationships:

<HCP, Registrar> and <Registrar, Consultant> for the classifier Role.

<TVAH, JCUH> and <TVAH, NTUH> for the classifier DataLocation.

<Admin, Finance> for the classifier DataType.

[8] TVAH = "Tees Valley Health Authority" HCP = "Health Care Professional" JCUH = "James Cook University Hospital" NTUH = "North Tees University Hospital"

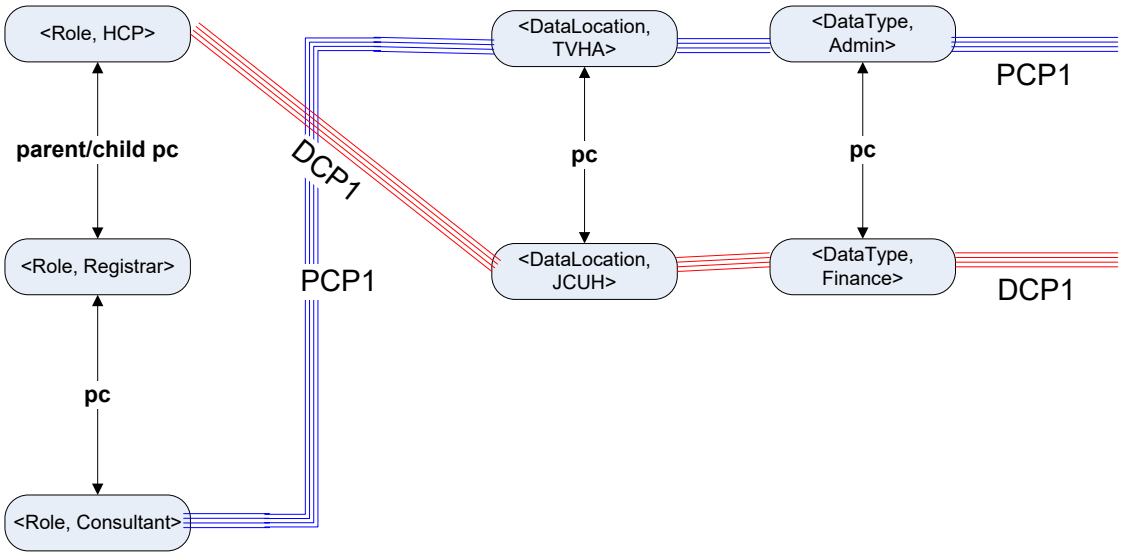


Figure 8: Confidentiality Permissions

Then for a consultant at North Tees University Hospital trying to access financial data at James Cook University Hospital the first match is PCP1 (where Role has the highest priority) and therefore access would be allowed. It is difficult to follow the logic that because Role has the highest priority PCP1 has preference over DCP1. DCP1 still contains a role i.e. HCP and this is meant to apply to all Roles including that of Consultant.

For TCM2 we look at first match in a slightly different way, and the benefits of this are:

1. We are not restricted to „limited hierarchies".
2. We use the concept of one CP refining another (3.2.3.3) which makes sense from a „real world" and an administrative point of view.
3. We do not need to rely on an ordering on the classifiers.

First match for TCM2 means that we look up all the inheritance paths for all the classifiers simultaneously. If a deny CP is found then the default applies that „deny overrules permit" so access is denied. If no deny CP is found and a permit CP is found then access is permitted.

So in the example above for a consultant at North Tees University Hospital trying to access financial data at James Cook University Hospital a deny CP would be found and so access would be denied. If for some peculiar reason we actually wanted to permit access then we would have to set up a permit CP of the form:

PCP2 = {<Role, Consultant>, <DataLocation, JCUH>, <DataType, Finance>}

PCP2 then refines DCP1 as described in 3.2.3.3. The CPs PCP1 and PCP2 (not DCP1) are found when applying the nearest match principle and so access is granted. An advantage of this type of refinement is that it can be verbally or visually described and a reason for it given e.g. „all health care professionals except consultants are denied access to financial data at James Cook University Hospital. Consultants are allowed access because …….. *whatever the reason is*"

Obviously, what is described above is a simplified and contrived example. The complexity in the original TCM increases significantly with the number of classifiers. However TCM2 fits in with the stated aim of „simplifying administration whilst at the same time providing sufficient granularity for the most complex „real life" authorisation scenarios" in a way that TCM1 does not, and also allows a generic specification using general hierarchies.

3.7 Specification in Spec Explorer

3.7.1 Introduction

Spec Explorer allows specification of variables and methods using sets and relations as in B. The project can be compiled producing an executable that can be run using test data. The methods can be linked, using a wrapper, to the methods in the actual application. Unlike B, the specification is not verified using a prover (although there is a static prover in Spec#). Instead, the specification is verified using random method calls on the full range of data. The specification here is applied in a limited way, as an introduction to how the different models compare. A full specification is developed when considering TCM3 (see 4.4).

3.7.2 Testing

The main method can call different scenarios, which allows the testing of data. In Scenario1 below I am testing (as part of development) the methods (operations in B) *AddUser*, *DeleteUser*, *AddCValue*, *DeleteCValue*, and *AssignUser*. I am also testing methods, which return the sets of user, operation, and object classifiers. Also, domain restriction (*domRes*) and relational composition (*relComp*) are tested. Relational Composition consists of three overridden methods to accommodate both functions (Maps) and relations. The variable *setone* tests relational composition.

3.7.3 Model

```
type USER = string;
```

```
type OP = string;
type OB = string;
type CFIER = string;
type VALUE = string;
type SESSION = string;

type CV = <CFIER,VALUE>;

type UA = <USER,CFIER,VALUE>;
type OPA = <OP,CFIER,VALUE>;
type OBA = <OB,CFIER,VALUE>;

type SV = <SESSION,CFIER,VALUE>;

var Set<CV> cvalues = Set{};
var Set<CFIER> cpts = Set{};
var Set<USER> users = Set{};
var Set<SV> session_values = Set{};
var Set<UA> ua = Set{};
var Set<OPA> opa = Set{};
var Set<OBA> oba = Set{};

Map<SESSION,USER> session_user = Map{};

invariant
Forall{<a,b,c> in session_values; <a,b,c> in relComp(session_user,ua)};
```

3.7.4 Main Method

A main method is needed for an executable model. The main method calls Scenario 1.

```
void Main()
{
  Scenario1();
}

[Action]
void AddUser(USER! user)
requires user notin users;
{
  users[user] = true;
}

[Action]
void DeleteUser(USER! user)
requires user in users;
{
  parallel
  {
  users[user] = false;
    foreach (<u,c,v> in ua)
    {
    ua[<user,c,v>] = false;
    }
    foreach (<s,u> in session_user, <t,c,f> in session_values, u ==
user, s==t)
    {
    session_values[<s,c,f>] = false;
```

```
    }
  }
}

[Action]
void AddCValue(CFIER cfier, VALUE val)
requires <cfier,val> notin cvalues;
{
  cvalues[<cfier,val>] = true;
}

[Action]
void DeleteCValue(CFIER cfier, VALUE val)
requires <cfier,val> in cvalues;{
  cvalues[<cfier,val>] = false;
  foreach (<u,c,f> in ua) {
    ua[<u,cfier,val>] = false;
  }
  foreach (<o,c,f> in oba) {
    oba[<o,cfier,val>] = false;
  }
  foreach (<o,c,f> in opa) {
    opa[<o,cfier,val>] = false;
  }
  foreach (<s,c,f> in session_values) {
    session_values[<s,cfier,val>] = false;
  }
}

[Action]
void AssignUser(USER user, CFIER cfier, VALUE val)
requires user in users && <cfier,val> in cvalues;
{
  ua[<user,cfier,val>] = true;
}

Set<<string,string>> relComp(Set<<string,string>> U,
Set<<string,string>> V) {
  return Set{<a,b> in U, <c,d> in V, b==c; <a,d>};
}

Set<<string,string>>relComp(Map<string,string> U,Set<<string,string>> V)
{
  return Set{<a,b> in U, <c,d> in V, b==c; <a,d>};
}

Set<<string,string,string>> relComp(Map<string,string> U,
Set<<string,string,string>> V) {
  return Set{<a,b> in U, <c,d,e> in V, b==c; <a,d,e>};
}

Set<CFIER> user_cfiers(Set<<USER,CFIER,VALUE>> UA) {
  return Set{<u,c,f> in UA; c};
}

Set<CFIER> op_cfiers(Set<<OP,CFIER,VALUE>> OPA) {
  return Set{<o,c,f> in OPA; c};
}
```

```
Set<CFIER> obj_cfiers(Set<<OB,CFIER,VALUE>> OBA) {
  return Set{<o,c,f> in OBA; c};
}

Set<<string,string>> domRes(Set<string> S, Set<<string,string>> R) {
  return Set{s in S, <x,y> in R, s==x; <x,y>};
}
```

3.7.5 Scenario

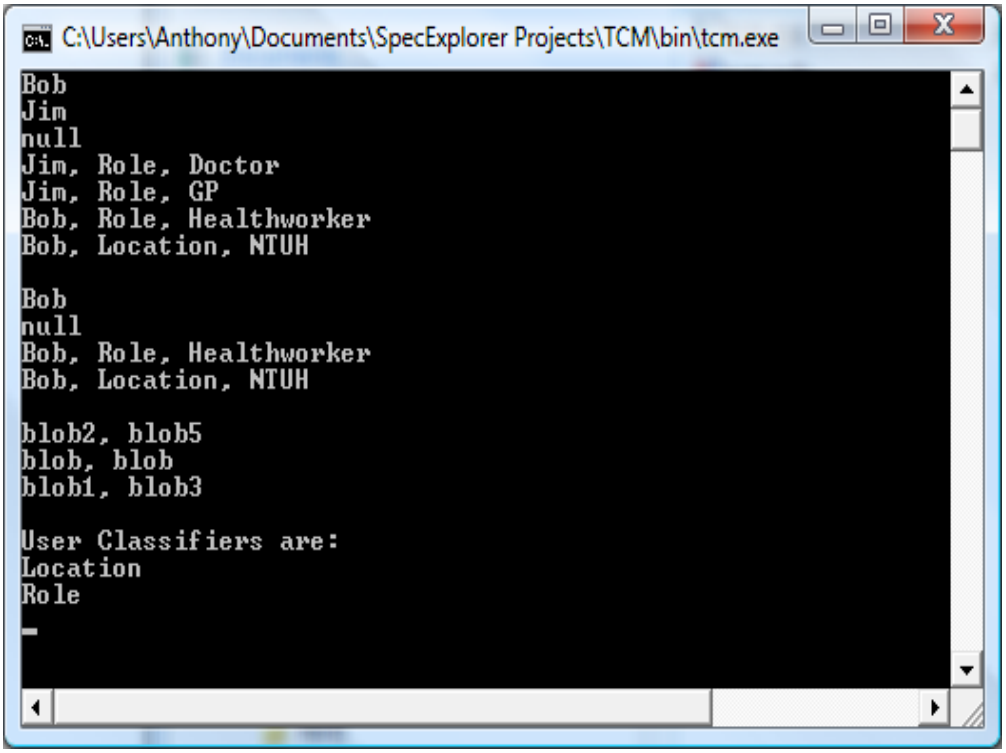
A scenario is a method that invokes model actions. A scenario method is itself an action of the model. However, unlike other kinds of actions, when a scenario calls other actions Spec Explorer records the intermediate states.

```
[Action(Kind=ActionAttributeKind.Scenario)]
void Scenario1()
{
  AddUser(null);
  AddCValue("","");
  AddUser("Jim");
  AddUser("Bob");
  AddCValue("Role","Doctor");
  AddCValue("Role","Healthworker");
  AddCValue("Role","GP");
  AddCValue("Location","NTUH");
  foreach (u in users)
  WriteLine(u);
  var i = 0;
  AssignUser("Jim","Role","Doctor");
  AssignUser("Jim","Role","GP");
  AssignUser("Bob","Role","Healthworker");
  AssignUser("Bob","Location","NTUH");
  foreach (s in ua)
  {
  let <a,b,c> = s;
  WriteLine(a + ", " + b + ", " + c);
  }
  ReadLine();
  DeleteUser("Jim");
  foreach (u in users)
  WriteLine(u);
  foreach (s in ua)
  {let <a,b,c> = s;
  WriteLine(a + ", " + b + ", " + c);}
  ReadLine();
  var Set<<string,string>> setone = Set{};
  setone[<"blob","blob">] = true;
  setone[<"blob1","blob2">] = true;
  setone[<"blob2","blob3">] = true;
  setone[<"blob3","blob5">] = true;
  setone = relComp(setone,setone);
  foreach (p in setone)
  {let <a,b> = p;
  WriteLine(a + ", " + b);}
  ReadLine();
```

```
  var Set<CFIER> uc = user_cfiers(ua);
  WriteLine("User Classifiers are: ");
 foreach (u in uc)
  WriteLine(u);
  ReadLine();
}
```

3.7.6 Output

The following is the output from the executable:



Console 2

3.8 Part-Application using SQL

A part application has been developed to demonstrate the use of the TCM. The B-Method can be translated directly into code written in C. This is not applicable here. However, the formal specification enables SQL to be written in a way that closely follows the B, giving the benefits of reusability as well as the benefits of consistency and verification normally associated with formal specification. In addition, the use of the CLR[9] in SQL Server 2005 may at some stage allow direct code translation.

None of this has applied to the code written for the TCM before this. Each demonstration has been written using a one-off „hacked" version. In addition, whilst this may be acceptable (and even beneficial) in non safety-critical applications, the use of formal

[9] CLR = "Common Language Runtime"

specification is essential in such a sensitive area as health service authorisation.

The part application discussed below and used as part of the development and testing of the TCM is more fully developed when used with TCM3 (6.10).

3.8.1 Example: B-Method and SQL

A given CP gives access for a particular user and data item according to the following expression:

CPPermitAccess (cp, acvals) ≔ **bool** (**dom** (acvals ∩ **ad** [cp]) = **dom** (cp))

For a particular instance, acvals is derived from the current user and the data item to which access is requested. So the above can be rewritten.

CPPermitAccess (cp, UserId, DataId) ≔
bool (**dom** (acvals (UserId, DataId) ∩ **ad** [cp]) = **dom** (cp))

The SQL that corresponds to this is below. Note that the parameters correspond, and note the use of the functionality in SQL of „intersect" and „except" which corresponds neatly to the B.

```
CREATE FUNCTION [dbo].[fnGivenCPPermitsAccess]
(@CPid int, @UserId sql variant = null, @DataId nvarchar(50) = null)
RETURNS int
AS

BEGIN
declare @output int
set @output = 0

declare  @intersect table (Classifier nvarchar(50),CValue nvarchar(50))
declare  @except table (Classifier nvarchar(50))


insert @intersect
select  Classifier,CValue from dbo.fnActiveCValues(@UserId,@DataId)
intersect
select Classifier,CValue from fnDescendants(@CPid)



insert @except
select Classifier from tcm PermitCPs p where p.CPid = @CPid
except
select Classifier from @intersect

IF NOT EXISTS
(select Classifier from @except)
```

```
SET @output = 1

RETURN @output
END
```

The function **fnActiveCValues(@UserId,@DataId)** gives the assigned classifier values, and **fnDescendants(@CPid)** corresponds to *ad* [*cp*] i.e. the classifier values of the given CP and all their descendants. The correlation, and the ability to translate from B to code (even though this is not done automatically) is apparent.

Similarly, the B operation *GivenCPTPermitsAccess* is written:

CPTPermitAccess (cpt, acvals) ≙
bool (∃cp. (cp ∈ permit_cps ∧**dom** (cp) = cpt ∧ **dom** (acvals ∩ **ad** [cp]) = cpt))

This can be rewritten as:

CPTPermitAccess (cpt, UserId, DataId) ≙
bool (∃cp. (cp ∈ permit_cps ∧ **dom** (cp) = cpt ∧
CPPermitAccess (cp, UserId, DataId)))

To program this in SQL firstly there is a SQL function to determine whether a given CP matches a given CPT i.e. to satisfy the expression **dom** (cp) = cpt. Again, the „intersect" and „except" functionality is used in SQL to match the B.

```
CREATE FUNCTION [dbo].[fnCPTtoCPmatch]
(@CPTid int,@CPid int)
RETURNS int
AS

BEGIN
declare @output int
set @output = 0

declare  @unionAB table (Classifier nvarchar(50))
declare  @intersectAB table (Classifier nvarchar(50))

insert @unionAB
select Classifier from tcm_CPT t where t.CPTid = @CPTid
union
select Classifier from tcm_PermitCPs p where p.CPid = @CPid

insert @intersectAB
select Classifier from tcm_PermitCPs p where p.CPid = @CPid
intersect
select Classifier from tcm CPT t where t.CPTid = @CPTid
```

```
IF NOT EXISTS
(select Classifier from @unionAB
except
select Classifier from @intersectAB)
SET @output = 1

RETURN @output
END
```

Using the function created above I can now write the function to determine whether a given CPT grants access as:

```
CREATE FUNCTION [dbo].[fnGivenCPTPermitsAccess]
(@CPTid int, @UserId sql_variant = null, @DataId nvarchar(50) = null)
RETURNS int
AS

BEGIN
declare @output int
set @output = 0

IF EXISTS
(select CPid from tcm PermitCPs
where [dbo].[fnCPTtoCPmatch](@CPTid,CPid)=1
and [dbo].[fnGivenCPPermitsAccess](CPid,@UserId,@DataId)=1)
SET @output = 1

RETURN @output
END
```

Similar programming in SQL can be done for all the B-method operations. It should be noted that the emerging technology of LinQtoSQL enables a much closer tie in between C# code and SQL, and thus a more obvious match to the B specification. In the final development of a TCM class, detailed in Section 6, which covers all versions of the TCM, only one basic stored procedure is used, in order to determine all the descendants of a CP. This becomes a method in LinQtoSQL. It is a possible area of further work, to determine which approach is optimal for large databases.

3.9 Summary

Following some discussion of the background of TCM concepts and the motivating health care scenario the TCM has been completely specified in B. The specification paralleled the RBAC specification with, of necessity, some increased complexity. The specification highlighted some areas where improvements could be made to the TCM and these were introduced, resulting in TCM2. Using the robust B specification as a basis, some specification was re-made in Spec Explorer as a means of animating the specification, and

as a step towards implementation in code. In addition, some SQL functions were created to show how they could be implemented in correspondence to the B.

4 TCM3

4.1 Introduction

The use of an authorisation model is a compromise between specifying every permission individually in an Access Control List (which was used in early models) and being able to simplify the administration whilst keeping sufficient granularity to apply the permissions as required. RBAC was a significant step along this path allowing collections of permissions to be given to collections of individuals based on role or inherited role. This greatly simplified administration and paralleled the „real world". However, in many cases the granularity has proved insufficient and occasioned the development of various add-ons to RBAC e.g. parameterised RBAC, which uses role plus some other factor, such as location as an additional parameter.

The TCM is another significant step. Essentially an authorisation model uses some form of collections to connect users, operations, and objects in order to simplify administration.

In RBAC, this is collections of users by role, and collections of permissions. In the TCM, this is generalised and „role" is not given a special place. In the TCM, collections are based on classifiers of which role is but one option. The TCM allows collections of operations and objects as well as users, and uses classifiers and classifier values to facilitate that.

The TCM as it has been presented in various papers and applied in practice is fully detailed in the previous section (3), and is shown to be an extension of RBAC. During the modelling of TCM2 questions arose such as:

- What does the model look like if we generalise it still further and allow any set of classifier values to be a CP? i.e. the CPs no longer have to conform to a fixed set of CPTs
- If we no longer have a fixed set of CPTs can we eliminate CPT (and Classifier) ordering and still retain the power of the model?

It could be said thatTCM3 is a step back from the implementation of TCM2 and consists of those elements of the TCM that generalise RBAC as a first step, without some of the additional features included in TCM2 such as fixed CPTs and CPT ordering.

From the research viewpoint in TCM3 we are considering what are the essential elements of an authorisation system that no longer has to have „role" as its central mechanism and

which allows not only multiple user classifiers to be used for authorisation, but also operation and object classifiers. TCM3 could be said to be a generic authorisation system that is generalisation of RBAC and is also a generalisation and simplication of TCM2.

It is the proposition for TCM3 that the use of different CPTs and CPT ordering in TCM2 adds a great deal of complexity to the model without the corresponding increase in useful granularity. The experience in the „real world" is that if authorisations are difficult for administrators to apply in their day-to-day work, they end up giving much wider authorisation than is really necessary. This defeats the increased granularity of a complicated model.

It is the proposition here that there is a use for a simplified TCM, which is called TCM3. TCM3 is a generalisation of RBAC and I believe it can be shown to be a great improvement over RBAC. Any authorisation requirement expressible in ordinary language can be applied e.g., "A member of an ambulance crew with a legitimate relation with the patient has access to all of a patient"s unsealed data. In an emergency situation, the crewmember has access to a patient"s sealed data. This access would be audited and justification required (at a later stage if necessary)". If we examine the above statement, it will be seen that we are simply applying authorisation to various collections. It is the proposition here that this simplified TCM simplifies administration through the use of collections whilst providing sufficient granularity for complex permissions – which RBAC with it"s insistence on collection by role does not.

A simplified TCM is presented along these lines together with examples of how this can be applied to real life scenarios. If required the features of TCM2 such fixed CPTs and CPT ordering can be added back in if required in a particular implementation. However, it is the proposition here that there is plenty of power in the simplified TCM without needing to do that.

4.2 TCM3 Reference Model

4.2.1 Re-examination of EHR Scenario

The EHR scenario detailed in 3.1.1 has been used in every presentation of the TCM. It has been used to justify the necessity for using different CPTs with an ordering on those CPTs. I wish now to re-examine these requirements with a view to applying a simplified TCM.

Firstly, these requirements require further discussion to ensure that the person setting the permissions (system administrator) has understood exactly what is wanted by the person requiring the permissions (health service manager?).

This is normal and would occur in a real-life situation, and acknowledges that there is a process of permissions‟ design that needs to be entered into. This scenario has, up until now, not been subject to that process. This is an essential stage in the process and needs to be done before meaningful discussion of how to apply the permissions. The requirements will now be restated with discussion.

1. *My GP, Fred, can see all my data.* Whether this permission is for Fred as an individual or in his role as a GP has been discussed in previous papers, and it has usually been decided to allocate the permission to Fred as an individual. However, does Fred really need the permission when he has retired and living in France? In most GP practices, there is shared practice responsibility amongst practice GPs. However, patients usually do have their own allocated GP even if they can see any GP. I would suggest that permissions could be given to any GP who has a legitimate relationship with the patient. If any permission was required to be given only to Fred then I would add the additional requirement that Fred was at that time a GP with a legitimate relationship with the patient.

2. *Nobody must know about my termination except my GP, any Gynaecological Consultant, and the Consultant Renal Transplant Surgeon who operated on me.* If nobody must know about this data then do not give anyone else permission to see this data. Or, does this mean nobody must *ever* know except the stipulated people. I would think that most system administrators would be reluctant to give up the right to change permissions when and if necessary. However, this can be done, and has been done with standard RBAC: some system administrators even removing their own rights from the system (not intentionally) at vast reinstall expense.

3. *My GP, Consultant Renal Transplant Surgeon, and Consultant Orthopaedic Surgeon can see my psychosis data, but no one else*. Once again, do not give the permission. If it is thought desirable to lock the permissions this can be done.

4. *I do not wish the members of the hospital team who carried out my termination operation to ever be able to see my psychosis data, except if they are viewing in a psychiatric role.* The requirement „ever‟ is a peculiar one. What if one of the team

became the patient‟s GP? Although this is unlikely, I think it is valid to question this admittedly contrived example. Are we saying that the system administrator cannot change permissions whatever happens? Leaving „ever‟ to the side for the moment, I presume there are a large number of other people who the patient would not want viewing the data. So perhaps it is better just to specify who can see the data and under what circumstances. I would think it not unreasonable that any psychiatrist with a legitimate relationship with the patient would be allowed to see the data, and this would cover the case given.

In summary, permissions design is a whole field of expertise in itself, one that has not been tackled in any great depth as regards the TCM, this being to do with Administration whereas we have largely concentrated on Authorisation. One of the jobs of any Administrator is to discuss in depth exactly what permissions will implement the correct requirements. I would expect such a discussion to occur if the EHR scenario was a real life scenario, and I would expect the points raised above to be part of that discussion. The job of the TCM is to make things as easy and useful as possible for the Administrator(s), otherwise they just end up giving far greater permissions than are actually necessary, with the corresponding security risks.

4.2.2 Authentication

Identity and access management systems consist of three main parts: authentication, authorisation, and administration. Techniques are being developed, in particular SAML, which enable single sign-on by allowing users to authenticate at an identity provider, and then access service providers without additional authentication. As far as authorisation is concerned, in TCM3 authentication is regarded as just another user classifier: the user is either authenticated or not, and a session is just that period of time during which the user is authenticated and interacting with the system. The expansion of Single Sign On and Federated Identity means that a user is authenticated if he has a token, or if he has a certificate, or if he is authenticated by another system and that system‟s authentication is recognized by the current system. This is consistent with the TCM3 view. The type or class SESSION is omitted from TCM3 as different from TCM2, a session just being regarded as the period of time during which a user is authenticated.

TCM3 uses the following sets (types or classes):

USER; OP; OB; CFIER; VALUE

The *users* of the system (machine or person) are of type *USER*.

$$users \subseteq USER$$

Classifier values are of type CVALUE, which is defined as:

$$CVALUE = \text{CFIER} \times VALUE$$

4.2.3 Assigned Classifier Values

The assigned classifier values (*acv*) are given by the following relation:

$$acv \in USER \times OP \times OB \leftrightarrow CVALUE$$

This relation can be written in the form below (and this has been found useful in practice):

$$acv = ua + opa + oba + uoba + svals^{10}$$

where

$$ua \in USER \leftrightarrow CVALUE$$
$$opa \in \text{OP} \leftrightarrow CVALUE$$
$$oba \in OB \leftrightarrow CVALUE$$

This is the normal user, operation, and object assignment used before in the TCM2 specification. The relation *uoba* is given by:

$$uoba \in USER \times OB \leftrightarrow CVALUE$$

This includes those classifier values such legitimate relationship, which depend on both the user and the object. Additionally there can be system values such as time or system condition, which are independent of user, operation and object.

$$svals \subseteq CVALUE$$

The full expression for *acv* is given in the footnotes. It is not used here because it does not (at present) correspond to any practical examples. However, the full expression is used for the Spec# specification (Section 4.4).

---

[10] The full expression would be acv = $ua + opa + oba + uoba + uopa + obopa + uobopa + svals$ where $uopa \in USER \times OP \leftrightarrow CVALUE$; $obopa \in OB \times OP \leftrightarrow CVALUE$; $uobopa \in USER \times OB \times OP \leftrightarrow CVALUE$

4.2.4 Permissions and Prohibitions

As permissions can be specified, using an access control list, which lists every user, operation, and object there is no essential need for prohibitions in an authorisation model except as and where it helps the administration. However it has proved useful e.g. in Microsoft‟s Active Directory, to have a prohibition which overrules all permissions. As a justification for this, one can imagine a situation in which the system administrator just wishes to deny access without any great study of interacting permissions in a permissions‟ design exercise.

Permissions design is a complex topic on its own, and permissions can be designed to give exactly what is required in fine detail. In TCM3, I apply the rule that any deny overrules all permits, but allow denies (and permits) to be refined. So the rule in TCM3 now becomes "Any unrefined deny overrules everything else". There are no CPTs (as used in TCM2) just permissions and prohibitions with the simple rule that an unrefined prohibition overrules all permissions.

It is the contention that this gives the best balance between granularity of permission and ease of administration. The administrator can add a prohibition and could then be immediately shown those (if any) permissions which refine that prohibition. These can then be removed or further refinements added at the appropriate level of granularity.

In TCM3, the rule that an unrefined prohibition overrules all permission applies irrespective of the permission type. In TCM2 permission at a higher level of CPT ordering is more important than a prohibition at lower level, and vice versa. Otherwise, in TCM2 permissions and prohibitions carry essentially the same weight, and interact within the CPT. This can make it difficult to see what the overall outcome of applying a permission or a prohibition is supposed to be.

Finally, even though TCM3 is a simplified version of TCM2 it is still a valuable generalisation of RBAC. It has a wider scope than RBAC and can be used to model current developments of RBAC e.g. parameterised RBAC and much more. The permissions and prohibitions are defined as below:

$$permit\_cps \subseteq \mathbb{F}\,(CVALUE)$$
$$deny\_cps \subseteq \mathbb{F}\,(CVALUE)$$

4.2.5 Inheritance

Inheritance is essential to TCM3 because the concept of deny CPs refining permit CPs (and vice versa) is essential to TCM3. Additionally the concept of more complex CPs (this will be explained later) refining less complex CPs is important to TCM3. Inheritance is implemented through a single classifier to parent to child relation.

$cpc \subseteq CFIER \times (VALUE \times VALUE)$

Specifying inheritance as a triple reflects the fact that inheritance is a partition on classifiers i.e. confined within roles, or locations, or data types etc., and is a development on the way it was defined in TCM2.

A classifier to ancestor to descendant relation is derived from the classifier to parent to child relation (cpc), as shown below.

$$cad \triangleq \{cfier, ad \mid cfier \in \text{CFIER} \land ad \in \text{VALUExVALUE} \land ad \in (\mathbf{ran}\ (\{cfier\} \lhd cpc))^*\}$$

This is the closure of the parent to child relation with each parent to child relation confined to its own classifier. As in TCM2, this relation includes that a value is counted as its own descendant.

A confidentiality permission grants (or denies) permission according to its classifier values and all the descendants of those classifier values. The definition **desc** is the set of all descendants of the classifier values in the confidentiality permission (x) defined as follows.

$$\begin{aligned}&\mathbf{desc}(x) \triangleq \{\text{cfier, childval} \mid \text{cfier} \in \text{CFIER} \land \text{childval} \in \text{VALUE} \land\\ &\quad \exists\ \text{parentval}.\ (\text{parentval} \in \text{VALUE} \land \text{cfier} \mapsto \text{parentval} \in x \land\\ &\quad (\text{parentval} \mapsto \text{childval} \in (\mathbf{ran}\ (\{\text{cfier}\} \lhd \text{cpc}))^*\}\end{aligned}$$

The definition gives the original classifier values plus all descendants of those values. Each subset of descendants is confined to its own classifier.

4.2.6 No CPT Ordering in TCM3

In TCM1 and TCM2, there is a set of CPTs on which there is an ordering. These CPTs effectively constrain the CPs that can be created and the ordering determines how the permissions given by each CPT are processed to give the overall permission. It could be said that in TCM3 there are no CPTs. However, it would be more truthful to say that there are no constraints so that every possible CPT is allowed. In addition, there is no ordering

on the CPTs and this removes much of the complexity of processing permissions. Every CP is equal except where one CP refines another (see 4.2.8 below).

4.2.7 No Classifier Ordering in TCM3

Classifier ordering in the original TCM was used for two reasons: firstly to enable a default ordering on the CPTs and secondly to determine which CP had preference within a CPT. This was because TCM1 adopted a nearest match policy for CPs within the same CPT. However, TCM2 modified this nearest match policy as discussed in 3.6. For example, suppose there is a GP Fred with a permitCP

PCP1 = {<Identity, Fred>, <Role, GP>, <PatientID, Alice>}

In addition, that for some reason Fred is banned from seeing records at the James Cook Hospital, and in effect, there exists the following denyCP for Alice"s records at the hospital

DCP1 = {<Identity, Fred>, <Location, JCUH>, <PatientID, Alice>}

The TCM1 approach would say that role is more important than location, therefore access is granted. TCM2 would say that these are two different permissions, one of which grants access while the other denies. Therefore the system default must operate, which is that deny overrules permit, therefore permission is denied. The TCM2 approach is continued to TCM3 with an extension to the concept of CP refinement, as discussed in the next section.

4.2.8 TCM3 Inheritance and Refinement

In TCM2, there is the case where one CP modifies or refines another CP. In TCM2, this modifying or refining is done within the same CPT through inheritance. For example the permit CP

PCP2 = {<Identity, Fred>, <Location, JCUH Outpatients>, <PatientID, Alice>}

refines DCP1 in the above paragraph where there is the inheritance triple <Location, JCUH, JCUH Outpatients> and grants Fred access in the outpatients department even though he is denied in JCUH as a whole.

It can be seen that DCP1 and PCP2 belong to the same CPT i.e. {Identity, Location, PatientId}

In TCM3 refinement is extended so that a CP can also be refined by another CP which belongs to more complex CPT e.g. the deny CP

DCP1 = {<Identity, Fred>, <Location, JCUH>, <PatientID, Alice>}

is refined by

PCP3 = {<Identity, Fred>, <Role, OutpatientsLocum>, <Location, JCUH>, <PatientID, Alice>}

so that although Fred is banned from seeing Alice"s data through his own identity, he is allowed if he is in the role of an outpatient"s session doctor.

For one CP *cp2* to refine another CP *cp1* in TCM3 there are two conditions that have to apply:

1. The CPT for *cp2* must be the same as, or include the CPT for *cp1* i.e.

   dom (*cp1*) ⊆ **dom** (*cp2*)

2. For the classifiers which *cp2* and *cp1* have in common (i.e. the classifiers of *cp1*) there must be classifier values of *cp1* higher up the inheritance tree than the classifier values of *cp2*. This is enforced by:

   dom (*cp1*) ◁ desc (cp2) ⊆ desc (cp1)

I believe that this is what is behind (in TCM1 and TCM2) the more general rule that the more complex CPTs take precedence over the less complex, but in TCM1 and TCM2 this is taken to apply to even disparate CPTs. In TCM3 CPs can refine other CPs as above. Otherwise the system default operates where CPs belong to disparate CPTs i.e. deny overrules permit. The B description for some CP to deny access for a set of assigned classifier values *acvals* is:

da ← **SomeCPDeniesAccess** (acvals) ≙
PRE
acvals ∈ F(cvalues)
THEN
da ≔ **bool** (∃ cp1. (cp1 ∈ deny_cps ∧
dom (acvals ∩ desc (cp1)) = **dom** (cp1) ∧
¬ (∃ cp2. (cp2 ∈ permit_cps ∧
dom (acvals ∩ desc (cp2)) = **dom** (cp2) ∧
dom (cp1) ⊆ **dom** (cp2) ∧
dom (cp1) ◁ desc (cp2) ⊆ desc (cp1)))))
END

This might seem complex but there are no further levels of complexity as in TCM1 and TCM2. The expression says that access is denied if there exists a CP *cp1* which denies access (either by itself or though inheritance) and there does not exists a CP *cp2* which refines *cp1* and permits access.

If the system default is „deny" this would be tested first, and if some CP denies access then the TCM as a whole denies access. (In keeping with the generic nature of the specification, it is easy to switch to a system default of „permit"). If there does not exist a CP that denies access then the following is evaluated.

pa ← **SomeCPPermitsAccess** (acvals) ≙
PRE
acvals ∈ F(cvalues)
THEN
pa :=**bool** (∃ cp1. (cp1 ∈ permit_cps ∧
dom (acvals ∩ desc (cp1)) = **dom** (cp1) ∧
¬ (∃ cp2. (cp2 ∈ deny_cps ∧
dom (acvals ∩ desc (cp2)) = **dom** (cp2) ∧
dom (cp1) ⊆ **dom** (cp2) ∧
dom (cp1) ◁ desc (cp2) ⊆ desc (cp1)))))
END

If the above returns true then the TCM permits access. If there is no CP to either permit or deny access then the system default operates which is usually that access is denied. This is given by:

pa ← **TCMPermitsAccess** (acvals) ≙
PRE
acvals ∈ F (cvalues)
THEN
pa := ¬ SomeCPDeniesAccess (acvals) ∧ SomeCPPermitsAccess (acvals)
END

4.2.9 Overrides

Overrides are widely used in the application of TCM1 and TCM2. The way they normally operate is that some CPTs are removed from the permission processing in an override situation. A widely used example is that data in a „sealed envelope" is accessible to certain individuals in certain situations. E.g., restrictions on access to the „sealed envelope" are removed in an „emergency" situation. Any access in this situation is strictly audited.

Essentially an override is just increased permissions in certain situations. From the viewpoint of TCM3, the „situation" can be regarded as a system classifier. Those people who can override can be specified using a user classifier; the data is classified as „in sealed envelope" and the condition can be specified as a system classifier value e.g. <situation, emergency>. Whatever the situation they will be some classifier for those who can override and they will be either able to use permissions normally denied or use new permissions in the override situation. This can be implemented using a prohibition of the form.

DCP = {<role, HCP>, <LegRel, Yes>, <Situation, Normal>, <DataCat, Sealed Envelope>}

which would prohibit access to sealed envelope data in any non-emergency situation, but allow any given permissions to operate in an emergency situation irrespective of whether they were in a sealed envelope or not. This could also be implemented using a permission

PCP = {<role, HCP>, <LegRel, Yes>, <Situation, Emergency>, <DataCat, Sealed Envelope >

However, this might grant someone permissions to which they were not normally entitled. Detailed permission design would depend on the actual implementation. However, the principle of treating an override situation as a classifier is clear.

4.2.10 Constrained TCM3

For TCM3 static separation of values is ignored, this not being used in many current RBAC applications e.g. Microsoft"s role provider and access control. In addition, the increased use of single sign on (SSO) and federated identity means that only dynamic separation of duties at runtime is considered.

As far as dynamic separation of values is concerned, constrained TCM can be regarded as just another confidentiality permission e.g. the deny CP

DCP = {<constrained values, yes>}

4.2.11 TCM3 Functional Specification Overview

For TCM3 the specification is not divided into Core, Hierarchical, and Constrained although some of the operations are specified without inheritance or refinement for development and testing purposes. The reason for not dividing TCM3 is that firstly inheritance and refinement is an essential part of TCM3, and secondly that a constraint can be regarded as just another classifier (see above).

4.2.12 Administrative Operations for TCM3

Administrators create and delete users, assign and deassign users to classifier values (which could include roles), and add or remove Confidentiality Permissions. Confidentiality Permissions give authorisation through user classifier values in combination with operation and object classifier values.

— AddUser: adds a user to the set *users*. Details of users are usually inputted before any authorisation is assigned, and may be kept on the system after they have left the company.

— DeleteUser: deletes a user.

— AssignUser: assigns a user to a classifier value. This could be a role, location, team, or any attribute the user could have.

— DeassignUser: deassigns a user from a classifier value.

— AddPermitCP: adds a Confidentiality Permission.

— RemovePermitCP: removes a Confidentiality Permission.

— AddDenyCP: adds a Confidentiality Prohibition.

— RemovePermitCP: removes a Confidentiality Prohibition

Additional administrative operations are concerned with the creation and maintenance of the classifier parent to child relationship *cpc*.

— AddInheritance: establishes a new immediate inheritance relationship between two classifier values.

— DeleteInheritance: deletes an existing immediate inheritance relationship between two classifier values.

4.2.13 Supporting System Operations for TCM3.

These operations are specified without inheritance and refinement, with inheritance but without refinement, and with both inheritance and refinement. It is only the last case that is used in a TCM3 implementation. The other cases are used for development and testing.

— GivenCPPermitsAccess: returns TRUE if the given CP permits access.

— GivenCPDeniesAccess: returns TRUE if the given CP denies access.

— SomeCPPermitsAccess: returns TRUE if some CP permits access.

— SomeCPDeniesAccess: returns TRUE if some CP denies access.

— SomeCPPermitsAccessI: returns TRUE if the given CP permits access through its own values or the descendants of those values.

— SomeCPDeniesAccessI: returns TRUE if the given CP denies access through its own values or the descendants of those values.

— SomeCPPermitsAccessIR: returns TRUE if the given CP permits access through its own values or the descendants of those values and the permission is not refined by a deny CP.

— SomeCPDeniesAccessIR: returns TRUE if the given CP denies access through its own values or the descendants of those values and the prohibition is not refined by a permit CP.

— TCMPermitsAccess: returns TRUE if the Tees Confidentiality Model overall permits access. Usually where SomeCPDeniesAccessIR is not true and SomeCPPermitsAccessIR is true, i.e. the system default is that "deny overrules permit".

4.2.14 Review Operations for TCM3

— AssignedUsers: returns the set of users assigned to a given classifier value.

— UserCValues: returns the set of classifier values assigned to a given user.

— AssignedCValues: returns the set of classifier values assigned for given user, operation, and object. These classifier values together with the CPs determine whether the operation is allowed on the object in the session.

— Permissions: returns the set of permissions available to a given user.

— OperationsOnObject: returns the set of operations that can be performed on the given object by a given user.

4.3 TCM3 Functional Specification in B

4.3.1 Administrative Operations

AddUser

This operation creates a new TCM user. The operation is valid only if the new user is of type *USER* and not already a member of *users*.

AddUser (user) ≙
  PRE
    user ∈ USER - users

THEN
user ≔ users ∪ {user}
END

DeleteUser

This operation deletes a user from the TCM. All user assignments are also deleted.

DeleteUser (user) ≙
PRE
user ∈ users
THEN
users ≔ users - {user} ||
acv := {user} x OP x OB ⩤ acv
END

AssignUser

This operation assigns a classifier value to a user. The operation is valid if the user is a member of *users* and the assignment does not already exist.

AssignUser (user, cfier, val) ≙
PRE
user ∈ users ∧cfier ∈ CFIER ∧val ∈ VALUE ∧
user ↦ (cfier ↦ val) ∉ ua
THEN
ua ≔ ua ∪ {user ↦ (cfier ↦ val)}
END

DeassignUser

This operation deletes the assignment of a classifier value to a user. The classifier value must be one of the user‟s assigned classifier values.

DeassignUser (user, cval) ≙
PRE
user ∈ users ∧
user ↦ cval ∈ ua
THEN
ua ≔ ua – {user ↦ cval}
END

AddPermitCP

This operation adds a Confidentiality Permission that acts to permit access. The CP consists of a set of classifier values and must not already be a CP.

AddPermitCP (cvals) ≙
PRE
cvals ∈ 𝔽(CVALUE) ∧
cvals ∉ permit_cps ∪ deny_cps
THEN
permit_cps ≔ permit_cps ∪ {cvals}
END

RemovePermitCP

This operation removes a permit CP, which by definition is a set of classifier values. The CP must already be a member of permit_cps.

RemovePermitCP (cvals) ≙
PRE
cvals ∈ permit_cps
THEN
permit_cps ≔ permit_cps - {cvals}
END

AddDenyCP

This operation adds a Confidentiality Prohibition, which acts to deny access. The CP consists of a set of classifier values and must not already be a CP.

AddDenyCP (cvals) ≙
PRE
cvals ∈ 𝔽(CVALUE) ∧
cvals ∉ permit_cps ∪ deny_cps
THEN
deny_cps ≔ deny_cps ∪ {cvals}
END

RemoveDenyCP

This operation removes a deny CP which by definition is a set of classifier values. The CP must already be a member of deny_cps.

RemoveDenyCP (cvals) ≙
PRE
cvals ∈ deny_cps

THEN
deny_cps ≔ deny_cps - {cvals}
END

AddInheritance

This operation establishes a new parent/ child relationship. The preconditions are that the relationship does not already exist and that cycle creation is avoided i.e. that the parent does not inherit from the child through the ancestor/ descendant relation ***ad***.

AddInheritance (cfier, pval, cval) ≙
PRE
cfier ∈ CFIER ∧ pval ∈ VALUE ∧ cval ∈ VALUE
cfier ↦ pval ∉ cvalues ∧ cfier ↦ cval ∉ cvalues
cfier ↦ (pval ↦ cval) ∉ cpc ∧
cfier ↦ (cval ↦ pval) ∉ cad
THEN
cpc ≔ cpc ∪ {cfier ↦ (pval ↦ cval)}
END

DeleteInheritance

This operation deletes a parent/ child relationship.

DeleteInheritance (cfier, pval, cval) ≙
PRE
cfier ∈ CFIER ∧ pval ∈ VALUE ∧ cval ∈ VALUE
THEN
cpc ≔ cpc - {cfier ↦ (pval ↦ cval)}
END

4.3.2 Supporting System Operations

GivenCPPermitsAccess

This operation returns true if the given CP permits access for the user, operation, and object with the assigned set of classifier values *acvals*. This is without inheritance or refinement.

pa ← **GivenCPPermitsAccess** (cp, acvals) ≙
PRE
cp ∈ permit_cps ∧ acvals ∈ 𝔽(CVALUE)
THEN
pa ≔ **bool** (**dom** (acvals ∩ cp) = **dom** (cp))

END

GivenCPDeniesAccess

This operation returns true if the given CP denies access for the user, operation, and object with the assigned set of classifier values *acvals*. This is without inheritance or refinement.

da ← **GivenCPDeniesAccess** (cp, acvals) ≙
PRE
cp ∈ deny_cps ∧ acvals ∈ 𝔽(CVALUE)
THEN
pa ≔ **bool** (**dom** (acvals ∩ cp) = **dom** (cp))
END

SomeCPPermitsAccess

This operation returns true if there exists a CP belonging to *permit_cps* that permits access for the user, operation, and object with the assigned set of classifier values *acvals*. This is without inheritance or refinement.

pa ← **SomeCPPermitsAccess** (acvals) ≙
PRE
acvals ∈ 𝔽(CVALUE)
THEN
pa ≔ **bool** (∃cp. (cp ∈ permit_cps ∧
(**dom** (acvals ∩ cp) = **dom** (cp)))
END

SomeCPDeniesAccess

This operation returns true if there exists a CP belonging to *deny_cps* that denies access for the user, operation, and object with the assigned set of classifier values *acvals*. This is without inheritance or refinement.

da ← **SomeCPDeniesAccess** (acvals) ≙
PRE
acvals ∈ 𝔽(CVALUE)
THEN
pa ≔**bool** (∃cp. (cp ∈ deny_cps ∧
(**dom** (acvals ∩ cp) = **dom** (cp)))
END

SomeCPPermitsAccessI

This operation returns true if there exists a CP belonging to *permit_cps* that permits access for the user, operation, and object with the assigned set of classifier values *acvals*. This is with inheritance but without refinement. The difference is that **desc**(*cp*) i.e. the set of all descendants of *cp* including *cp* itself, is used for authorisation instead of just *cp* itself.

pa ← **SomeCPPermitsAccessI** (acvals) ≙
PRE
acvals ∈ 𝔽(CVALUE)
THEN
pa ≔ **bool** (∃cp. (cp ∈ permit_cps ∧
(**dom** (acvals ∩ **desc** (cp)) = **dom** (cp)))
END

SomeCPDeniesAccessI

This operation returns true if there exists a CP belonging to *deny_cps* that denies access to the user, operation, and object with the assigned set of classifier values *acvals*. This is with inheritance but without refinement. The difference is that **desc**(*cp*) i.e. the set of all descendants of *cp* including *cp* itself, is used for authorisation instead of just *cp* itself.

da ← **SomeCPDeniesAccess** (acvals) ≙
PRE
acvals ∈ 𝔽(CVALUE)
THEN
pa ≔**bool** (∃cp. (cp ∈ deny_cps ∧
(**dom** (acvals ∩ **desc** (cp)) = **dom** (cp)))
END

SomeCPPermitsAccessIR

This operation returns true if there exists a CP belonging to *permit_cps* that permits access for the user, operation, and object with the assigned set of classifier values *acvals*. This is with inheritance and with refinement. For a detailed explanation of how inheritance and refinement work in TCM3 see 4.2.8.

pa ← **SomeCPPermitsAccessIR** (acvals) ≙
PRE
acvals ∈ 𝔽(CVALUE)
THEN
pa ≔**bool** (∃cp1. (cp1 ∈ permit_cps ∧
dom (acvals ∩ desc (cp1)) = **dom** (cp1) ∧

¬ (∃cp2. (cp2 ∈ deny_cps ∧
dom (acvals ∩ desc (cp2)) = **dom** (cp2) ∧
dom (cp1) ⊆ **dom** (cp2) ∧
dom (cp1) ◁ desc (cp2) ⊆ desc (cp1)))))
END

SomeCPDeniesAccessIR

This operation returns true if there exists a CP belonging to *deny_cps* that denies access for the user, operation, and object with the assigned set of classifier values *acvals*. This is with inheritance and with refinement. For a detailed explanation of how inheritance and refinement work in TCM3 see 4.2.8.

da ← **SomeCPDeniesAccessIR** (acvals) ≙
PRE
acvals ∈ F(CVALUE)
THEN
da :=**bool** (∃cp1. (cp1 ∈ deny_cps ∧
dom (acvals ∩ desc (cp1)) = **dom** (cp1) ∧
¬ (∃cp2. (cp2 ∈ permit_cps ∧
dom (acvals ∩ desc (cp2)) = **dom** (cp2) ∧
dom (cp1) ⊆ **dom** (cp2) ∧
dom (cp1) ◁ desc (cp2) ⊆ desc (cp1)))))
END

TCMPermitsAccess

This operation returns true if the TCM overall permits access. Access is permitted if and only if it is not denied by some CP (with inheritance and refinement) and is also permitted by some CP (with inheritance and refinement).

pa ← **TCMPermitsAccess** (acvals) ≙
PRE
acvals ∈ F(CVALUE)
THEN
pa := ¬ SomeCPDeniesAccessIR (acvals) ∧
SomeCPPermitsAccessIR (acvals)
END

4.3.3 Review Operations

AssignedCValues

This operation returns the assigned classifier values for a given *user, op, ob*. It is these values that are used for authorisation together with the CPs.

acvals ← **AssignedCVValues** (user, op, ob) ≙
PRE
user ∈ USER ∧ op ∈ OP ∧ ob ∈ OB
THEN
acvals ≔ **acv** [{user ↦op ↦ob}]
END

Permissions

This operation returns the permissions granted to a given *user.*

prms ← **Permissions** (user) ≙
PRE
user ∈ USER
THEN
prms ≔ {op, ob | op ∈ OP ∧ ob ∈ OB ∧
TCMPermitsAccess (**acv** [{user ↦op ↦ob}])
END

OperationsOnObject

This operation returns the operations allowed for a given user on a given object.

ooos ←**OperationsOnObject** (user, ob) ≙
PRE
user ∈ USER ∧ ob ∈ OB
THEN
ooos ≔ {op | op ∈ OP ∧ TCMPermitsAccess (**acv** [{user ↦op ↦ob}])
END

4.4 TCM3 Functional Specification in Spec#

4.4.1 Introduction

This section contains a Spec# model of TCM3. This is a mixture of actual code that can be run, and comments. The code is written in a special style and so only the code is compiled.

4.4.2 System State

The similarities to the B specification can be seen in that there are a number of types and variables. Variables are written as tuples, so there is an immediate correspondence to working with a database. The methods (operations) are written using the standard C#

method format with the addition of "requires" which corresponds to the precondition in B. The similarity to the B specification is such that comments on the code have been kept to a minimum. Writing a specification in Spec# while using the B specification as a basis allows the specification to be animated, and is a step towards final implementation.

```
type USER = string!;
type OP = string!;
type OB = string!;
type CFIER = string!;
type VALUE = string!;

type CVALUE = <CFIER,VALUE>;

type CP = Set<<CFIER,VALUE>>;
```

As in all specifications, there is a set of users.

```
var Set<USER> users = Set{};
```

The relations *ua*, *opa*, *oba* represent the assignment of users, operations and objects to their corresponding classifier values.

```
var Set<<USER,CFIER,VALUE>> ua = Set{};
var Set<<OP,CFIER,VALUE>> opa = Set{};
var Set<<OB,CFIER,VALUE>> oba = Set{};
```

Some classifier values are dependent on both user and object e.g. whether there is a legitimate relationship between the user and object, or the relative location of user and object. These are modelled by the relation *uoba*.

```
var Set<<USER,OB,CFIER,VALUE>> uoba = Set{};
```

Other classifier values are independent of user, operation, object e.g. whether there is an emergency situation, or what the time is. These are modelled by the variable *svals*.

```
var Set<<CFIER,VALUE>> svals = Set{};
```

For completeness, I include relations that model cases where classifier values are dependent on user and operation, operation and object, and where dependent on user, operation, and object together.

```
var Set<<USER,OP,CFIER,VALUE>> uopa = Set{};
var Set<<OP,OB,CFIER,VALUE>> opba = Set{};
var Set<<USER,OP,OB,CFIER,VALUE>> uopba = Set{};
```

In a TCM implementation, the data for all the above relations would be stored in tables. Only those tables useful for a particular application would be created. In the case studies so far tables for the above three relations have not been necessary. However, there may be

applications where they are needed. Having defined the type CP as a set of classifier values, permit_cps and deny_cps are defined as below.

```
var Set<CP> permit_cps = Set{};
var Set<CP> deny_cps = Set{};
```

The inheritance relation is defined as a set of triples.

```
var Set<<CFIER,VALUE,VALUE>> cpc = Set{};
```

4.4.3 Administrative Methods

4.4.3.1 A new user is added

```
[Action]
void AddUser(USER! user)
requires user notin users;
{
  users[user] = true;
}
```

4.4.3.2 A user is deleted

```
[Action]
void DeleteUser(USER! user)
requires user in users;
{
  parallel
  {
  users[user] = false;
    foreach (<u,c,v> in ua)
    {
    ua[<user,c,v>] = false;
    }
    foreach (<u,b,c,v> in uoba)
    {
    uoba[<user,b,c,v>] = false;
    }
    foreach (<u,p,b,c,v> in uopba)
    {
    uopba[<user,p,b,c,v>] = false;
    }
  }
}
```

4.4.3.3 A user is assigned to a classifier value.

```
[Action]
void AssignUser(USER! user, CFIER! cfier, VALUE val)
requires user in users && ua[<user,cfier,val>] == false;
{
  ua[<user,cfier,val>] = true;
}
```

4.4.3.4 A user is deassigned from a classifier value.

```
[Action]
void DeassignUser(USER! user, CFIER! cfier, VALUE val)
requires user in users && <user,cfier,val> in ua;
{
  ua[<user,cfier,val>] = false;
}
```

4.4.3.5 A permit CP is created.

```
[Action]
void AddPermitCP(CP pcp)
requires pcp notin permit_cps;
requires pcp notin deny_cps;
{
  permit_cps[pcp] = true;
}
```

4.4.3.6 A permit CP is removed.

```
[Action]
void RemovePermitCP(CP pcp)
requires pcp in permit_cps;
{
  permit_cps[pcp] = false;
}
```

4.4.3.7 A deny CP is created.

```
[Action]
void AddDenyCP(CP dcp)
requires dcp notin permit_cps;
requires dcp notin deny_cps;
{
  deny_cps[dcp] = true;
}
```

4.4.3.8 A deny CP is removed.

```
[Action]
void RemoveDenyCP(CP dcp)
requires dcp in deny_cps;
{
  deny_cps[dcp] = false;
}
```

4.4.3.9 An inheritance relation is added.

The added classifier-parent-child triple must not already exist and must not create a cycle.

```
[Action]
void AddInheritance(CFIER! cfier, VALUE pval, VALUE cval)
requires <cfier,pval,cval> notin cpc &&
<cfier,cval,pval> notin closure(cpc);
{
```

```
  cpc[<cfier,pval,cval>] = true;
}
```

4.4.3.10 An inheritance relation is deleted.

```
[Action]
void DeleteInheritance(CFIER! cfier, VALUE pval, VALUE cval)
requires <cfier,pval,cval> in cpc;
{
  cpc[<cfier,pval,cval>] = false;
}
```

4.4.4 System Methods

4.4.4.1 Without Inheritance: Given CP permits access

This method and the next three methods are as they would be without inheritance or refinement. They were used for testing in the development of the model. Because inheritance and refinement are essential to TCM3, they are not used in the final model. This is shown by the absence of the [Action] attribute.

```
bool  PermitAccessNoIR(CP pcp, USER! user, OP! op, OB! ob)
requires pcp in permit_cps && user in users;
{
  return dom(AssignedCValues(user,op,ob)*pcp)==dom(pcp);
}
```

4.4.4.2 Without Inheritance: Given CP denies access

This is without inheritance or refinement.

```
bool  DenyAccessNoIR(CP dcp, USER! user, OP! op, OB! ob)
requires dcp in deny_cps && user in users;
{
  return dom(AssignedCValues(user,op,ob)*dcp)==dom(dcp);
}
```

4.4.4.3 Without Inheritance: Some CP permits access.

This is without inheritance or refinement.

```
bool  PermitAccessNoIR(USER! user, OP! op, OB! ob)
requires user in users;
{
  return Exists{cp in permit_cps; PermitAccessNoIR(cp,user,op,ob)};
}
```

4.4.4.4 Without Inheritance: Some CP denies access.

This is without inheritance or refinement.

```
bool  DenyAccessNoIR(USER! user, OP! op, OB! ob)
requires user in users;
{
```

```
  return Exists{cp in deny_cps; DenyAccessNoIR(cp,user,op,ob)};
}
```

4.4.4.5 With Inheritance: Given CP permits access

This method and the three methods below return true if a given CP or the CPs as a whole permit or deny access through their own values or any inherited values. Permit CPs are not refined by Deny CPs or vice versa. Similar to the B specification, these were used for testing in the development of the model. They are not used in the final model, as shown by the absence of the [Action] attribute.

```
bool  PermitAccessNoR(CP pcp, USER! user, OP! op, OB! ob)
requires pcp in permit_cps && user in users;
{
  return dom(AssignedCValues(user,op,ob)*closure(pcp,cpc))==dom(pcp);
}
```

4.4.4.6 With Inheritance: Given CP denies access

This is with inheritance, but without refinement.

```
bool  DenyAccessNoR(CP dcp, USER! user, OP! op, OB! ob)
requires dcp in deny_cps && user in users;
{
  return dom(AssignedCValues(user,op,ob)*closure(dcp,cpc))==dom(dcp);
}
```

4.4.4.7 With Inheritance: Some CP permits access

This is with inheritance, but without refinement.

```
bool  PermitAccessNoR(USER! user, OP! op, OB! ob)
requires user in users;
{
  return Exists{cp in permit_cps; PermitAccessNoR(cp,user,op,ob)};
}
```

4.4.4.8 With Inheritance: Some CP denies access

This is with inheritance, but without refinement.

```
bool  DenyAccessNoR(USER! user, OP! op, OB! ob)
requires user in users;
{
  return Exists{cp in deny_cps; DenyAccessNoR(cp,user,op,ob)};
}
```

4.4.4.9 With Refinement: Given CP permits access

This is with inheritance and refinement. Used in the model. The given CP permits access if there is no refinement by a Deny CP to deny access. The specification parallels the B specification (4.3.2).

```
[Action]
bool  PermitAccess(CP pcp, USER! user, OP! op, OB! ob)
requires pcp in permit_cps && user in users;
{
  return dom(AssignedCValues(user,op,ob)*closure(pcp,cpc))==dom(pcp) &&
  !Exists{dcp in deny_cps;
  dom(AssignedCValues(user,op,ob)*closure(dcp,cpc))==dom(dcp) &&
  dom(pcp) <= dom(dcp) &&
  domRestriction(dom(pcp),descendants(dcp)) < descendants(pcp)
  };
}
```

4.4.4.10 With Refinement: Given CP denies access

This is with inheritance and refinement. Used in the model. The specification parallels the B specification 4.4.4.10.

```
[Action]
bool  DenyAccess(CP dcp, USER! user, OP! op, OB! ob)
requires dcp in deny_cps && user in users;
{
  return dom(AssignedCValues(user,op,ob)*dcp)==dom(dcp) &&
  !Exists{pcp in permit_cps;
  dom(AssignedCValues(user,op,ob)*closure(pcp,cpc))==dom(pcp)&&
  dom(dcp) <= dom(pcp) &&
  domRestriction(dom(dcp),descendants(pcp)) < descendants(dcp)
  };
}
```

4.4.4.11 With Refinement: Some CP permits access

This is with inheritance and refinement. Used in the model. The specification parallels the B specification (4.3.2).

```
[Action]
bool  PermitAccess(USER! user, OP! op, OB! ob)
requires user in users;
{
  return Exists{pcp in permit_cps; PermitAccess(pcp,user,op,ob)};
}
```

4.4.4.12 With Refinement: Some CP denies access

This is with inheritance and refinement. Used in the model. The specification parallels the B specification (4.3.2).

```
[Action]
bool  DenyAccess(USER! user, OP! op, OB! ob)
requires user in users;
{
  return Exists{dcp in deny_cps; DenyAccess(dcp,user,op,ob)};
}
```

The TCM permits access depending on whether the system default is "permit overrules deny" or vice versa. Following the more usual practice that deny overrules everything, the method for the TCM permitting access is given as follows:

```
[Action]
bool  TCMPermitAccess(USER! user, OP! op, OB! ob)
requires user in users;
{
  return !DenyAccess(user,op,ob) && PermitAccess(user,op,ob);
}
```

4.4.5 Review Methods

4.4.5.1 Users

This simply returns the set of users.

```
[Action]
Set<USER> ViewUsers()
{
  return
      Set{u in users; u};
}
```

4.4.5.2 Permissions

This method returns the set of permissions obtained by the given user.

```
[Action]
Set<<OP,OB>> Permissions(USER! user)
{
  return
       Set{u in users, p in Operations(), b in Objects(),
TCMPermitAccess(u,p,b),u==user;<p,b>};
}
```

4.4.5.3 Operations on Object

This method returns the set of operations allowed by the given user on the given object.

```
[Action]
Set<OP> OperationsOnObject(USER! user, OB! ob)
{
  return
      Set{<u,p,b> in Set<<USER,OP,OB>>,
      TCMPermitAccess(u,p,b),u==user,b==ob;p};
}
```

4.4.6 Definitions

4.4.6.1 AssignedCValues

The method below returns the set of assigned classifier values related to a particular user, operation, object, and used for authorisation. As stated previously some of the terms may not be necessary in a particular application.

```
Set<<CFIER,VALUE>> AssignedCValues(USER! user, OP! op, OB! ob)
requires user in users;
{
  return

        Set{<u,c,v> in ua, u==user; <c,v>} +
        Set{<p,c,v> in opa, p==op; <c,v>} +
        Set{<b,c,v> in oba, b==ob; <c,v>} +

        Set{<u,b,c,v> in uoba, u==user,b==ob; <c,v>} +
        Set{<u,p,c,v> in uopa, u==user,p==op; <c,v>} +
        Set{<p,b,c,v> in opba, p==op,b==ob; <c,v>} +

        Set{<u,p,b,c,v> in uopba, u==user,p==op,b==ob; <c,v>} +

        svals;
}
```

4.4.6.2 Operations

This simply returns the set of operations

```
Set<OP> Operations()
{
  return

        Set{<p,c,v> in opa; p} +
        Set{<u,p,c,v> in uopa; p} +
        Set{<p,b,c,v> in opba; p} +
        Set{<u,p,b,c,v> in uopba; p};
}
```

4.4.6.3 Objects

This simply returns the set of objects

```
Set<OB> Objects()
{
  return

        Set{<b,c,v> in oba; b} +
        Set{<u,b,c,v> in uoba; b} +
        Set{<p,b,c,v> in opba; b} +
        Set{<u,p,b,c,v> in uopba; b};
}
```

4.4.6.4 Descendants

The set of all descendants of the classifier values in a confidentiality permission, including the original classifier values of the confidentiality permission. This gives a new set of classifier values that are themselves a confidentiality permission, and can be used to give permission (or prohibition) through inheritance.

```
Set<<CFIER,VALUE>> descendants(Set<<CFIER,VALUE>> CP)
{
  return  closure(CP,cpc);
}
```

4.4.7 Relational Operators

A number of relational operators, that came by default with the B Toolkit, had to be explicitly written in Spec#. This enabled the Spec# specification to follow, and take advantage of, previous work done in in the B specification. These relational operators, such as Domain and Range are given in Appendix B.

4.4.8 Main Method

You need a main method for an executable model. It can be used to call different scenarios. The scenarios are used during model development, to check the model‟s correct working. The third scenario is being called here.

```
void Main() {
  Scenario3();
}
```

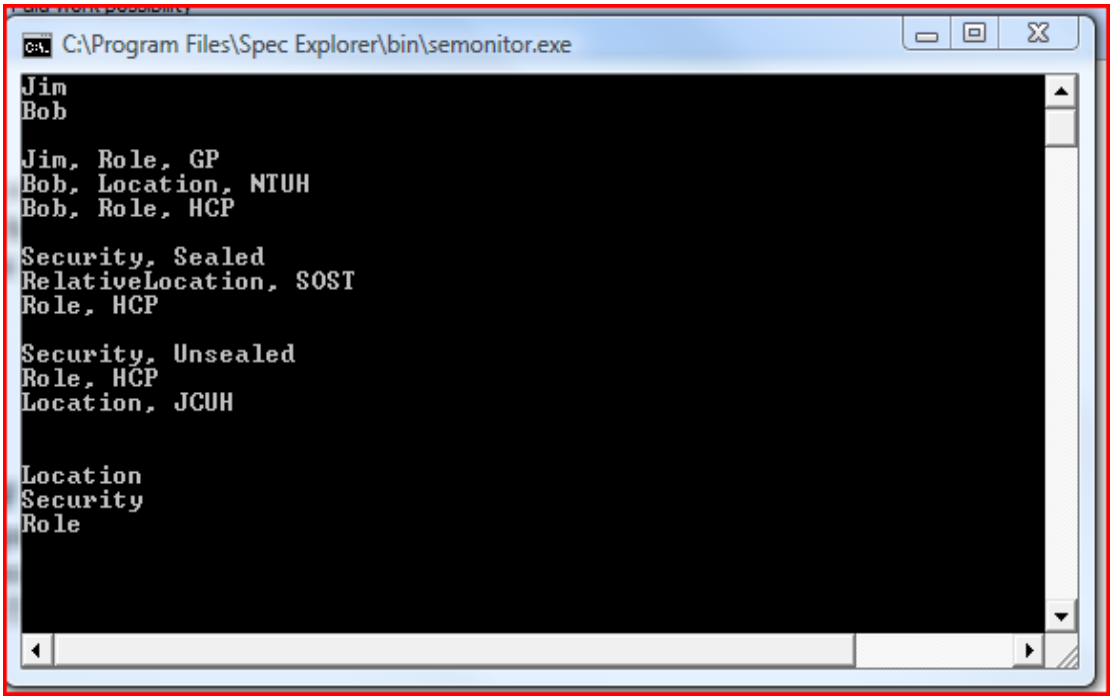
4.4.8.1 Scenario 1

This first scenario adds users, assigns classifier values, creates two permit cps, and outputs the cps and the domain of one of them.

```
[Action(Kind=ActionAttributeKind.Scenario)]
void Scenario1()
{
  AddUser(“Jim”);
  AddUser(“Bob”);
  foreach (u in users)
  WriteLine(u);
  WriteLine(“”);

  AssignUser(“Jim”,”Role”,”GP”);
  AssignUser(“Bob”,”Role”,”HCP”);
  AssignUser(“Bob”,”Location”,”NTUH”);
  foreach (s in ua)
  {
  let <a,b,c> = s;
  WriteLine(a + “, “ + b + “, “ + c);
```

```
  }
  WriteLine("");

  var Set<<CFIER,VALUE>> pcp1 = Set{};
  pcp1[<"Role","HCP">] = true;
  pcp1[<"Location","JCUH">] = true;
  pcp1[<"Security","Unsealed">] = true;
  AddPermitCP(pcp1);

  var Set<<CFIER,VALUE>> pcp2 = Set{};
  pcp2[<"Role","HCP">] = true;
  pcp2[<"RelativeLocation","SOST">] = true;
  pcp2[<"Security","Sealed">] = true;
  AddPermitCP(pcp2);

  foreach (p in permit_cps)
  {
  foreach (cv in p)
  {let <a,b> = cv;
  WriteLine(a + ", " + b);}
  WriteLine("");}
  WriteLine("");

  foreach (c in dom(pcp1))
  WriteLine(c);
  ReadLine();

  RemovePermitCP(pcp1);
  RemovePermitCP(pcp2);
  DeassignUser("Jim","Role","GP");
  DeassignUser("Bob","Role","HCP");
  DeassignUser("Bob","Location","NTUH");
  DeleteUser("Jim");
  DeleteUser("Bob");
}
```

The output from this Scenario 1 is shown in Console 3 below:



Console 3

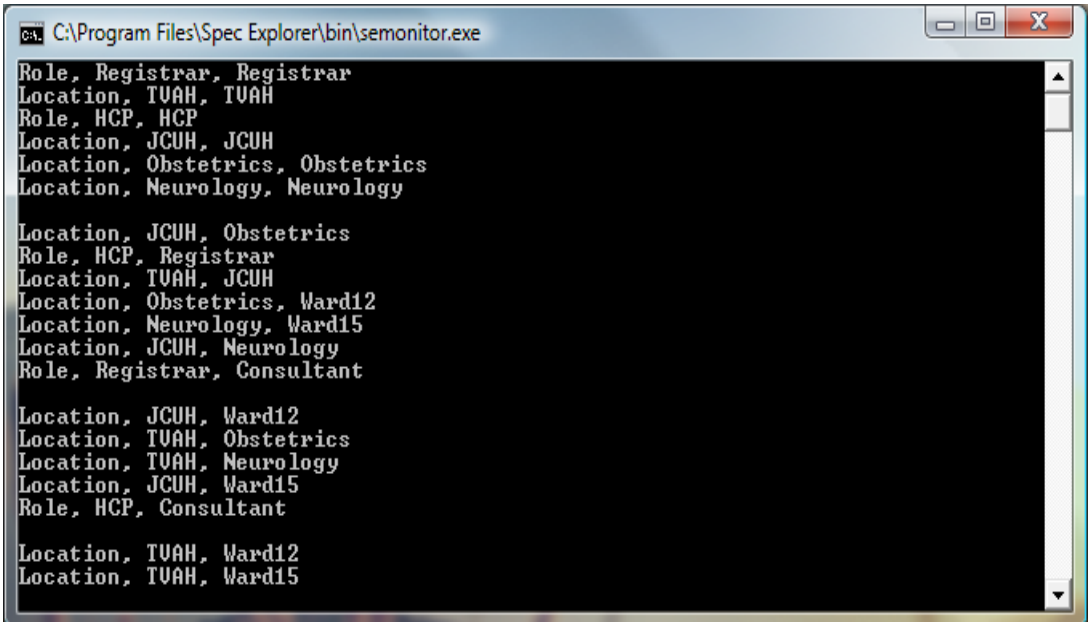
4.4.8.2 Scenario 2

This scenario tests iterates, closure, and descendants.

```
[Action(Kind=ActionAttributeKind.Scenario)]
void Scenario2()
{
  cpc[<"Role","HCP","Registrar">] = true;
  cpc[<"Role","Registrar","Consultant">] = true;
  cpc[<"Location","TVAH","JCUH">] = true;
  cpc[<"Location","JCUH","Obstetrics">] = true;
  cpc[<"Location","Obstetrics","Ward12">] = true;
  cpc[<"Location","JCUH","Neurology">] = true;
  cpc[<"Location","Neurology","Ward15">] = true;

var Set<<CFIER,VALUE,VALUE>> res0 = iterate(cpc,0);
  foreach (r in res0)
  {
  let <a,b,c> = r;
  WriteLine(a + ", " + b + ", " + c);
  }
  WriteLine("");

var Set<<CFIER,VALUE,VALUE>> res1 = iterate(cpc,1);
  foreach (r in res1)
  {
  let <a,b,c> = r;
  WriteLine(a + ", " + b + ", " + c);
  }
  WriteLine("");

var Set<<CFIER,VALUE,VALUE>> res2 = iterate(cpc,2);
  foreach (r in res2)
```

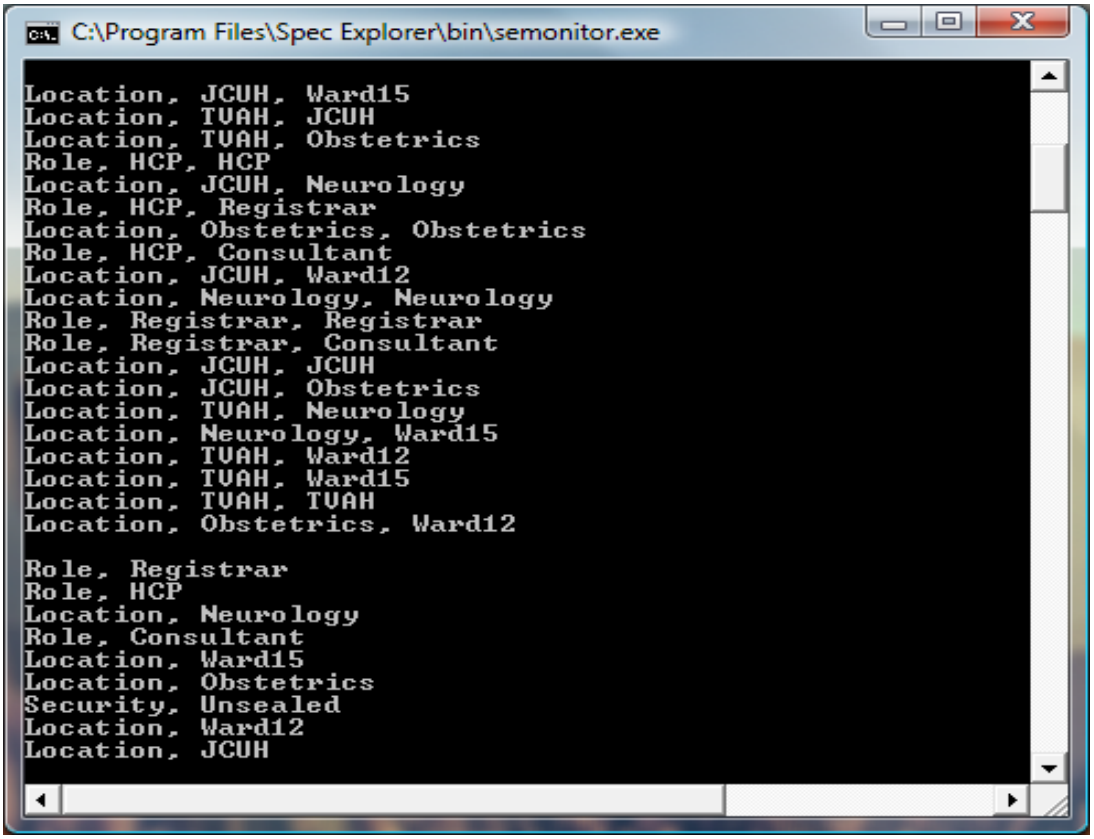
```
  {
  let <a,b,c> = r;
  WriteLine(a + “, “ + b + “, “ + c);
  }
  WriteLine(“”);

  var Set<<CFIER,VALUE,VALUE>> res3 = iterate(cpc,3);
  foreach (r in res3)
  {
  let <a,b,c> = r;
  WriteLine(a + “, “ + b + “, “ + c);
  }
  WriteLine(“”);



  var Set<<CFIER,VALUE,VALUE>> clos = closure(cpc);
  foreach (r in clos)
  {
  let <a,b,c> = r;
  WriteLine(a + “, “ + b + “, “ + c);
  }
  WriteLine(“”);

  var Set<<CFIER,VALUE>> pcp = Set{};
  pcp[<”Role”,”HCP”>] = true;
  pcp[<”Location”,”JCUH”>] = true;
  pcp[<”Security”,”Unsealed”>] = true;
  AddPermitCP(pcp);
  var Set<<CFIER,VALUE>> desc = descendants(pcp);
  foreach (d in desc)
  {
  let <a,b> = d;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);



  ReadLine();

}
```

The output of the first four iterates is shown in Console 4 below.



Console 4

The output for closure and descendants is shown in Console 5 below.



Console 5

4.4.8.3 Scenario 3

Alice has sealed and locked her data at North Tees University Hospital (NTUH) that refers to her termination. Consultant Jim at NTUH can see the data, although he is warned that it

is sensitive. Consultant Bob at James Cook University Hospital (JCUH) cannot see the data, and indeed is unaware that the data exists. Alice is referred for an operation at JCUH and confides in consultant Bob, who feels it is important that the surgical team have access to the data. There are two possibilities: Alice can downgrade the security of her data to just „sealed" which would mean that any other Health Care Professional could view the data (although this access would be audited), or the prohibition could be refined (with Alice"s explicit written consent) to allow access by the surgical team. Refinement is explored in this scenario.

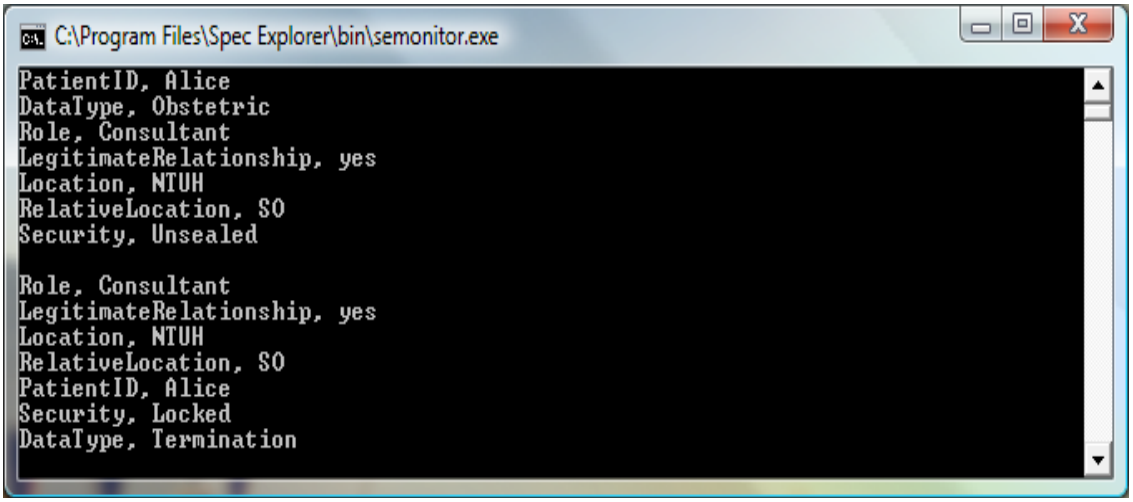
```
[Action(Kind=ActionAttributeKind.Scenario)]
void Scenario3()
{
  AddUser("Jim");
  AddUser("Bob");

  //User Classifier Values
  AssignUser("Jim","Role","Consultant");
  AssignUser("Bob","Role","Consultant");
  AssignUser("Jim","Location","NTUH");
  AssignUser("Bob","Location","JCUH");

  //single operation classified by its own identity.
  opa[<"read","OpID","read">]=true;

  //Object Classifier Values for some of Alice's Electronic Health
Records
  oba[<"EHR1","PatientID","Alice">]=true;
  oba[<"EHR1","DataType","Obstetric">]=true;
  oba[<"EHR1","Security","Unsealed">]=true;
  oba[<"EHR11","PatientID","Alice">]=true;
  oba[<"EHR11","DataType","Termination">]=true;
  oba[<"EHR11","Security","Locked">]=true;

  //Inheritance Relationships
  cpc[<"Role","HCP","Registrar">] = true;
  cpc[<"Role","Registrar","Consultant">] = true;
  cpc[<"DataType","Obstetric","Termination">] = true;

  //Classifier Values which depend on both user and object
  uoba[<"Jim","EHR1","LegitimateRelationship","yes">]=true;
  uoba[<"Jim","EHR11","LegitimateRelationship","yes">]=true;
  uoba[<"Jim","EHR1","RelativeLocation","SO">]=true;
  uoba[<"Jim","EHR11","RelativeLocation","SO">]=true;

WriteLine("Assigned CValues Jim,EHR1,EHR2");
  var Set<<CFIER,VALUE>> acvals = AssignedCValues("Jim","Read","EHR1");
  foreach (r in acvals)
  {
  let <a,b> = r;
  WriteLine(a + ", " + b);
  }
  WriteLine("");
```

```
acvals = AssignedCValues("Jim","Read","EHR11");
foreach (r in acvals)
{
let <a,b> = r;
WriteLine(a + ", " + b);
}
WriteLine("");


WriteLine("Results of Boolean Methods");
//Create Permit CP
var Set<<CFIER,VALUE>> pcp1 = Set{};
pcp1[<"Role","HCP">] = true;
pcp1[<"LegitimateRelationship","yes">] = true;
pcp1[<"Security","Unsealed">] = true;
AddPermitCP(pcp1);

//Create Permit CP. SO = 'Same Organisation'
var Set<<CFIER,VALUE>> pcp2 = Set{};
pcp2[<"Role","HCP">] = true;
pcp2 [<"LegitimateRelationship","yes">] = true;
pcp2 [<"RelativeLocation","SO">] = true;
pcp2 [<"Security","Locked">] = true;
AddPermitCP(pcp2);

//Create Deny CP. DO = 'Different Organisation'
var Set<<CFIER,VALUE>> dcp1 = Set{};
dcp1 [<"Role","HCP">] = true;
dcp1 [<"RelativeLocation","DO">] = true;
dcp1 [<"Security","Locked">] = true;
AddDenyCP(dcp1);


WriteLine(PermitAccessNoIR(pcp1,"Jim","Read","EHR1"));
WriteLine(PermitAccessNoR(pcp1,"Jim","Read","EHR1"));
WriteLine(PermitAccess(pcp1,"Jim","Read","EHR1"));

WriteLine(PermitAccessNoIR("Jim","Read","EHR1"));
WriteLine(PermitAccessNoR("Jim","Read","EHR1"));
WriteLine(PermitAccess("Jim","Read","EHR1"));

WriteLine(DenyAccessNoIR(dcp1,"Jim","Read","EHR1"));
WriteLine(DenyAccessNoR(dcp1,"Jim","Read","EHR1"));
WriteLine(DenyAccess(dcp1,"Jim","Read","EHR1"));

WriteLine(DenyAccessNoIR("Jim","Read","EHR1"));
WriteLine(DenyAccessNoR("Jim","Read","EHR1"));
WriteLine(DenyAccess("Jim","Read","EHR1"));
WriteLine("");
WriteLine("TCMPermitAccess");
WriteLine(TCMPermitAccess("Jim","Read","EHR1"));

WriteLine("Jim and Bob's Permissions before referral to Bob");
WriteLine("Jim's Permissions");
var Set<<OP,OB>> jprms = Permissions("Jim");
foreach (p in jprms)
{
```

```
  let <a,b> = p;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);

  WriteLine(“Bob’s Permissions”);
  var Set<<OP,OB>> bprms = Permissions(“Bob”);
  foreach (p in bprms)
  {
  let <a,b> = p;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);

//Referral to consultant Bob at JCUH
  uoba[<”Bob”,”EHR1”,”LegitimateRelationship”,”yes”>]=true;
  uoba[<”Bob”,”EHR11”,”LegitimateRelationship”,”yes”>]=true;
  uoba[<”Bob”,”EHR1”,”RelativeLocation”,”DO”>]=true;
  uoba[<”Bob”,”EHR11”,”RelativeLocation”,”DO”>]=true;

WriteLine(“Assigned CValues Bob,EHR1,EHR2”);
  acvals = AssignedCValues(“Bob”,”Read”,”EHR1”);
  foreach (r in acvals)
  {
  let <a,b> = r;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);

  acvals = AssignedCValues(“Bob”,”Read”,”EHR11”);
  foreach (r in acvals)
  {
  let <a,b> = r;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);

WriteLine(“Jim and Bob’s Permissions after referral to Bob”);
WriteLine(“Jim’s Permissions”);
  jprms = Permissions(“Jim”);
  foreach (p in jprms)
  {
  let <a,b> = p;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);

  WriteLine(“Bob’s Permissions”);
  bprms = Permissions(“Bob”);
  foreach (p in bprms)
  {
  let <a,b> = p;
  WriteLine(a + “, “ + b);
  }
  WriteLine(“”);


AssignUser(“Bob”,”Team”,”Surgical1”);
```

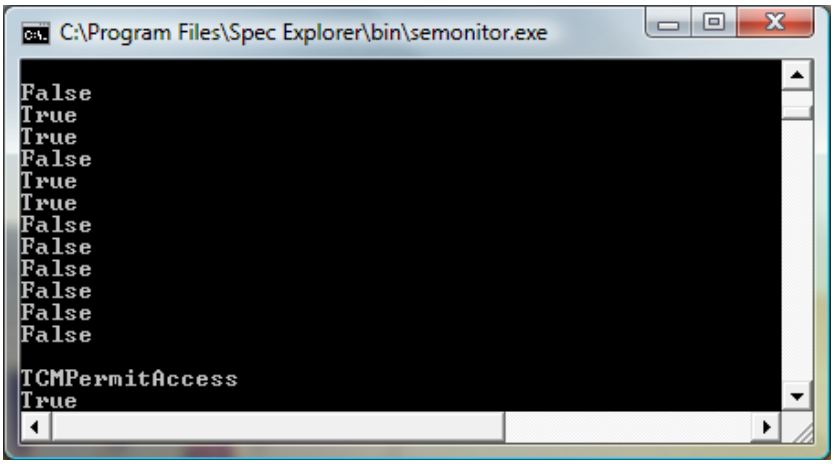
```
// Add a new permit CP which refines a deny CP.
var Set<<CFIER,VALUE>> pcp3 = Set{};
  pcp3 [<"Role","HCP">] = true;
  pcp3 [<"RelativeLocation","DO">] = true;
  pcp3 [<"Security","Locked">] = true;
  pcp3 [<"Team","Surgical1">] =true;
  AddPermitCP(pcp3);


WriteLine("Jim and Bob's Permissions after Permission Refinement");
WriteLine("Jim's Permissions");
  jprms = Permissions("Jim");
  foreach (p in jprms)
  {
  let <a,b> = p;
  WriteLine(a + ", " + b);
  }
  WriteLine("");

  WriteLine("Bob's Permissions");
  bprms = Permissions("Bob");
  foreach (p in bprms)
  {
  let <a,b> = p;
  WriteLine(a + ", " + b);
  }
  WriteLine("");









  DeassignUser("Jim","Role","Consultant");
  DeassignUser("Bob","Role","Consultant");
  DeassignUser("Bob","Location","NTUH");
  DeleteUser("Jim");
  DeleteUser("Bob");
}
```

Console 6 below shows the set of assigned classifer values for Consultant Jim and Electronic Health Record EHR1, followed by the set of assigned classifer values for Consultant Jim and Electronic Health Record EHR11.



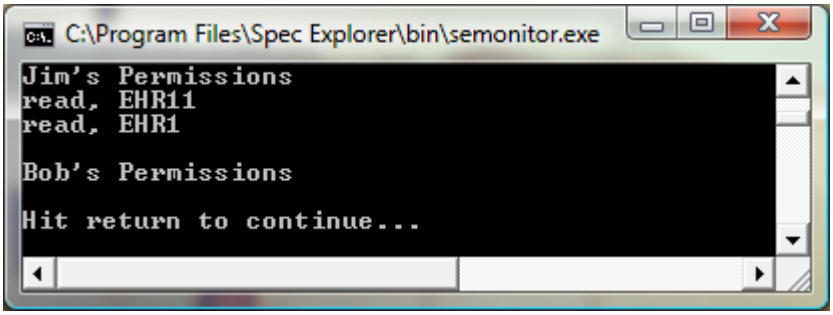
Console 6

The Boolean values returned by the various "Permit Access" expressions in the scenario are shown in Console 7.



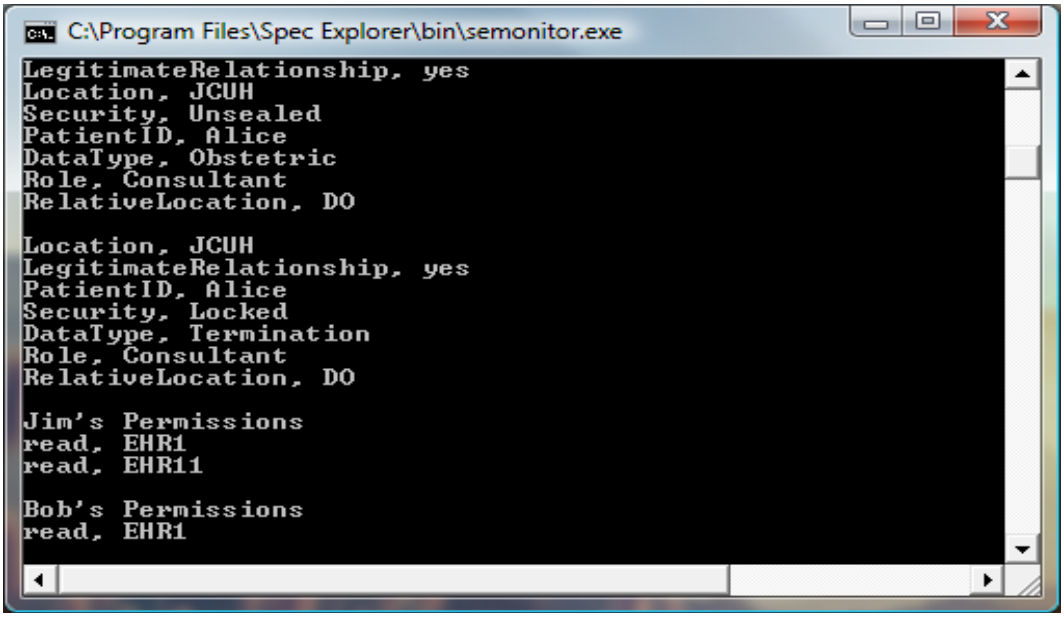
Console 7

Console 8 shows Jim and Bob"s permissions before referral to Bob.



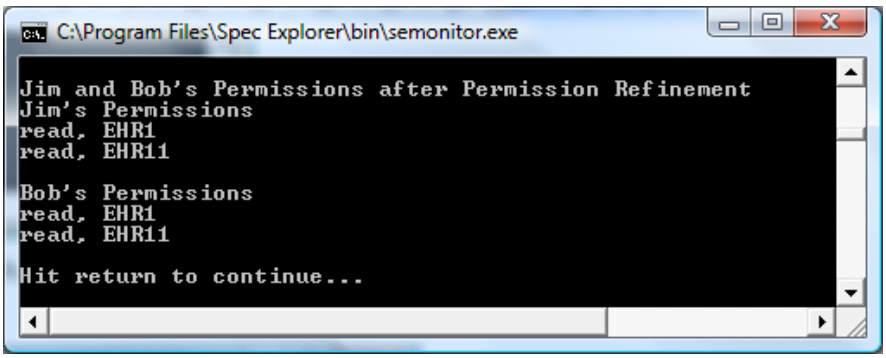
Console 8

Console 9 shows the assigned classifier values for Bob & EHR1, followed by the assigned classifier values for Bob & EHR11. The permissions for Jim and Bob are shown after referral to Bob i.e. when Bob has a legitimate relationship with Alice.



Console 9

The permissions for Jim are now shown (Console 10) after refinement of the Confidentiality Permision denying access, to allow access to Surgical Team 1 of which Bob is a member.

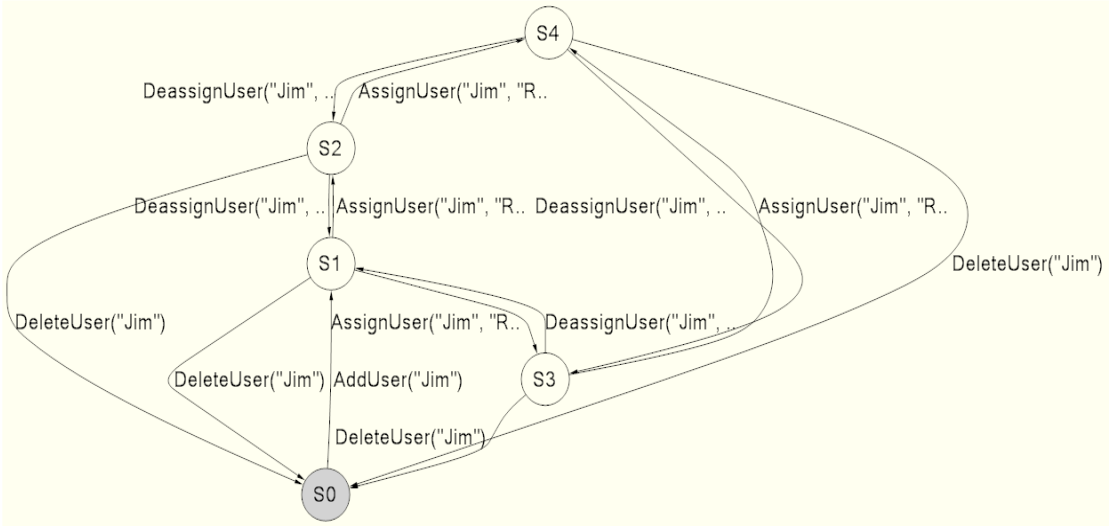


Console 10

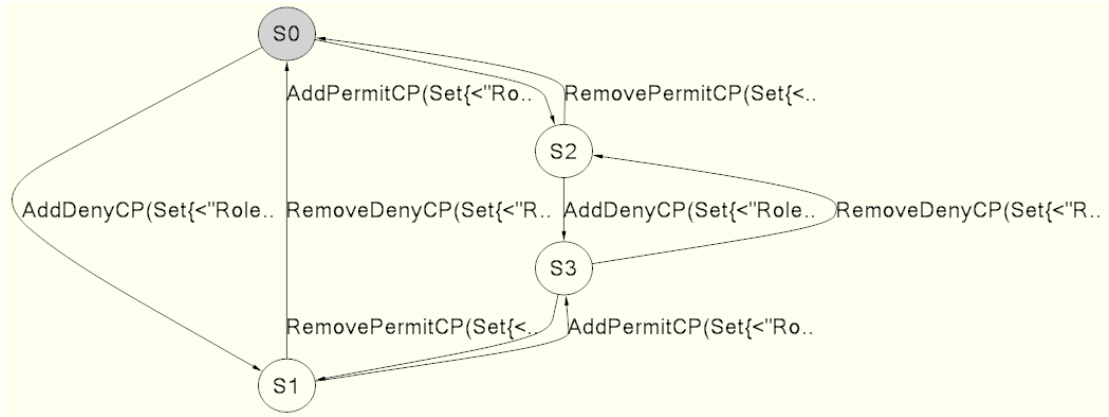
4.4.9 Exploration

Spec Explorer includes an exploration feature. The output consists of possible runs of the model program that it discovers. Spec Explorer represents this data as a finite-state machine [FSM]. The nodes of the FSM are the states of the model program before and after the invocation of a top-level method (an action); the transitions or links of the FSM represent method invocations, including arguments and return values. Some small parts of the FSM are shown graphically below. In reality, they can all be run together, and are better viewed through the SpecExplorer program. The FSMs are used to generate test suites for the implementation.

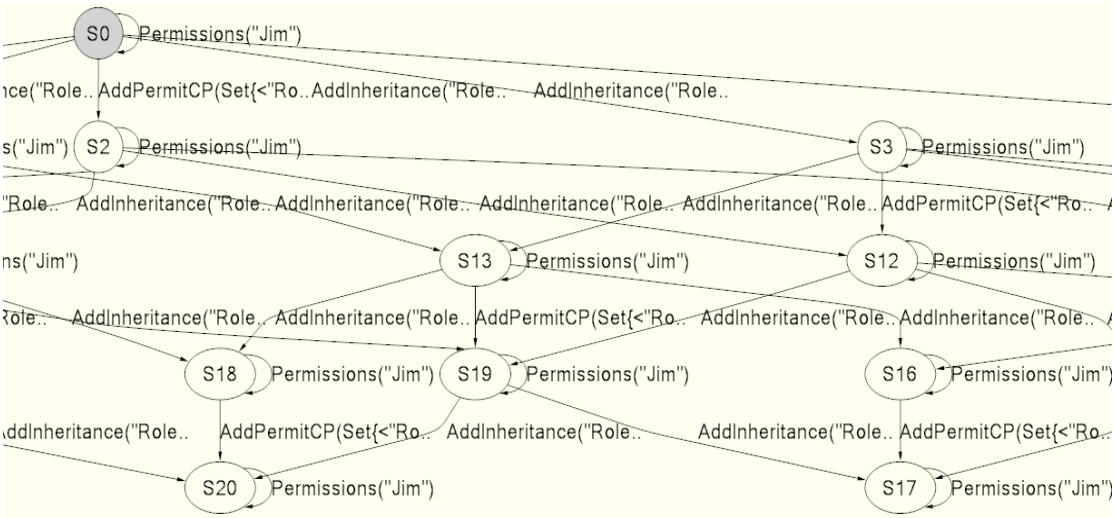
4.4.9.1 AddUser, DeleteUser, AssignUser, DeassignUser



4.4.9.2 AddPermitCP, RemovePermitCP, AddDenyCP, RemoveDenyCP.



4.4.9.3 AddPermitCP, AddInheritance, Permissions.



4.4.10 Test Suites

FSMs may be used to produce one or more test suites, which can be run against the implementation of the system. An API driver program can be written to make this easier. The purpose of the API driver is to do whatever is necessary in order to invoke the individual actions of the test suites in the implementation under test. In the simplest case, this is just a pass-through call to another .NET assembly. The application test suites to a TCM class are shown after presentation of the TCM class in the next section.

4.5 Comparison of Spec# with the B-Method

It has to be acknowledged that creating the model in B first, made it easier to rewrite it in Spec#. However, I can see no major difference as regards creating the model once the different syntaxes have been understood.

One advantage of the B-Method is that the model in B can be verified; whilst in Spec# it can only be explored using finite state machines. However, verification in B can require considerable effort even in quite simple models. It is only because of the suitability of RBAC and the TCM to specification in B, that it was not too hard to do it in these cases.

An advantage of Spec# is that, used in SpecExplorer, Spec# has all the features of an object-oriented language: classes, structs, overloading, polymorphism, enumerations etc. The model can then be used for testing with any object-oriented language by writing harnesses to the implementation. Of course, it works best with C#. The B Method as used in this research is not properly object oriented.

The Spec# code as written in SpecExplorer, using sets and relations, can be compiled and run. The B-Method model can be refined to an implementation in C. However, most modern application are unlikely to be written in C, so it may be that the „harness“ approach is best.

The way the B Method and Spec# have been used together in this research is that the initial specification has been done in B, which allows the specification to be verified. The B has then been used to write the specification in Spec Explorer, which allows the model to be animated. This is also a stage towards final implementation and allows the final implementation to be tested using harnesses created in Spec Explorer.

4.6 Translation to SQL

4.6.1 Descendants

It is possible to use the B specification to guide the SQL as shown previously. The introduction of CTE (Common Table Expressions) in SQL Server 2005 makes it easier to write recursive expressions. The following SQL uses the WITH clause to produce a set of descendants from a given CP (including the original ancestors):

```
WITH descs (Classifier,CValue)
AS
(SELECT Classifier, CValue from tcm_CPs
WHERE CPid = '11'
UNION ALL
SELECT  i.Classifier, i.Child from Inheritance i
      INNER JOIN descs on
            descs.Classifier = i.Classifier
AND         descs.CValue = i.Parent
)
SELECT DISTINCT * FROM descs
```

This is quite neat for a recursive function in SQL – certainly considerably neater than before. As an example, if we have the Confidentiality Permission below where CPid = „11":

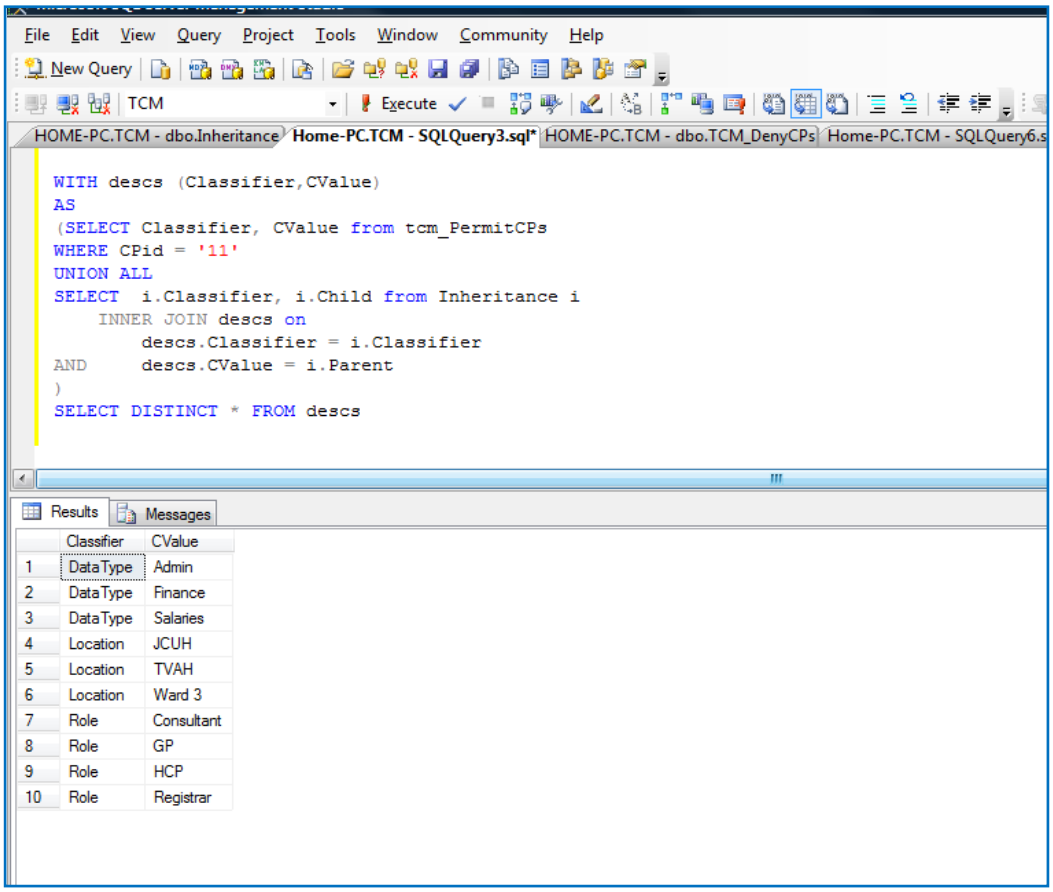
CP = {<Role, HCP>, <Location, TVAH>, <DataType, Admin>}

Together with the following inheritance table:

Classifier	Parent	Child
Role	HCP	GP
Role	HCP	Registrar
Role	Registrar	Consultant
Location	TVHA	JCUH
Location	JCUH	Ward 3
DataType	Admin	Finance
DataType	Finance	Salaries

Table 1: SQL Inheritance

Then the output from the above SQL is:



Screen 1: Descendants

A "descendants" function can then be used as part of the specification of the main authorisation (with inheritance) functions for TCM3 written in SQL (as follows).

4.6.2 Given CP Permits Access

```
CREATE FUNCTION [dbo].[fnGivenCPPermitsAccess]
(@CPid int, @UserId sql_variant = null, @DataId nvarchar(50) = null)
RETURNS int
AS

BEGIN
declare @output int
set @output = 0

declare  @intersect table (Classifier nvarchar(50),CValue nvarchar(50))
declare  @except table (Classifier nvarchar(50))

insert @intersect
select  Classifier,CValue from dbo.fnActiveCValues(@UserId,@DataId)
intersect
select Classifier,CValue from fnDescendants(@CPid)

insert @except
select Classifier from tcm_PermitCPs p where p.CPid = @CPid
except
select Classifier from @intersect
```

```
IF NOT EXISTS
(select Classifier from @except)

SET @output = 1

RETURN @output
END
```

4.6.3 Some CP Permits Access

```
CREATE FUNCTION [dbo].[fnSomeCPPermitsAccess]
(@UserId sql_variant = null, @DataId nvarchar(50) = null)
RETURNS int
AS

BEGIN
declare @output int
set @output = 0

IF  EXISTS
(select CPid from TCM_PermitCPs
where dbo.fnGivenCPPermitsAccess(CPid,@UserId,@DataId)=1)
SET @output = 1

RETURN @output
END
```

The SQL functions for GivenCPDeniesAccess and SomeCPDeniesAccess are almost exact copies of the functions above with permit replaced by deny.

4.6.4 TCM Permits Access

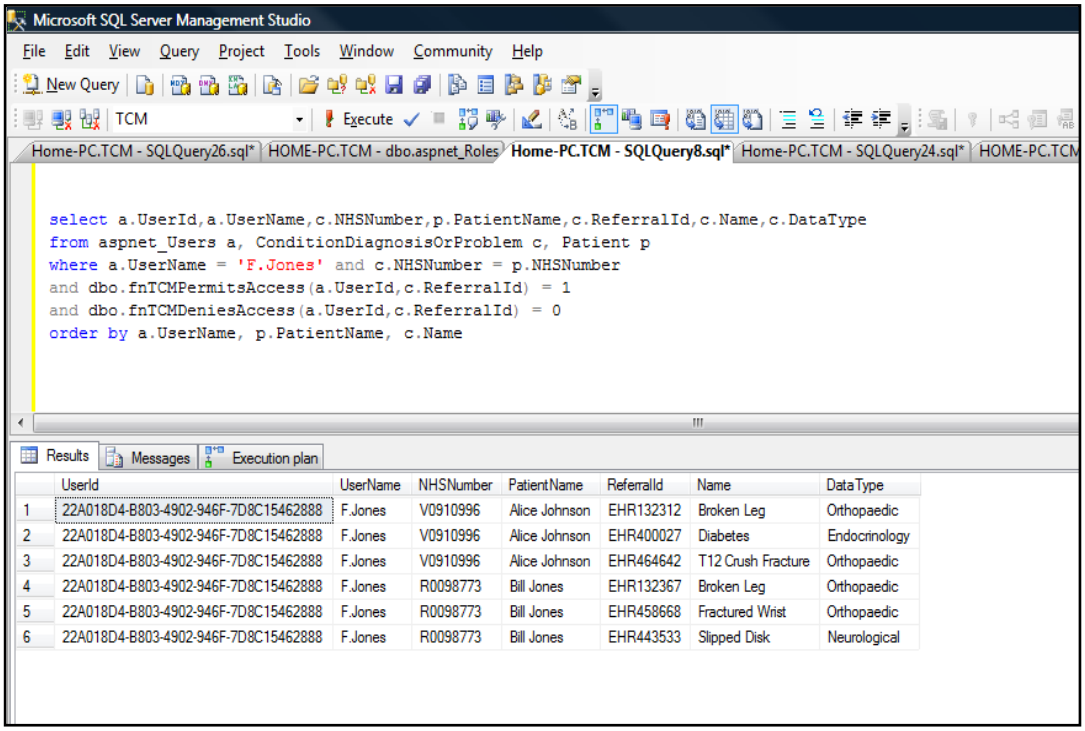
The SQL function below determines whether the TCM overall permits access. It closely follows the B specification.

```
CREATE FUNCTION [dbo].[fnTCMPermitsAccess]
(@UserId sql_variant = null, @DataId nvarchar(50) = null)
RETURNS int
AS

BEGIN
declare @output int
set @output = 0

IF EXISTS
(select * from TCM_PermitCPs p
where dbo.fnGivenCPPermitsAccess(p.CPid,@UserId,@DataId)=1
AND NOT EXISTS
(select * from TCM_DenyCPs d
where dbo.fnGivenCPDeniesAccess(d.CPid,@UserId,@DataId)=1
AND NOT EXISTS
(select Classifier from TCM_PermitCPs where CPid = p.CPid
except
select Classifier from TCM_DenyCPs where CPid = d.CPid)
AND NOT EXISTS
(select * from dbo.fnDescendants(d.CPid) r
```

```
where r.Classifier IN (select Classifier from TCM_PermitCPs where CPid =
p.CPid)
except
select * from dbo.fnDescendants(p.CPid))))
SET @output = 1

RETURN @output
END
```

4.6.5 Query Results

The query below shows what is accessible by user F.Jones. He is allowed to see these records because of his role as a Health Care Professional (HCP), and because the data is classified for security purposes as „Unsealed"



	UserId	UserName	NHSNumber	PatientName	ReferralId	Name	DataType
1	22A018D4-B803-4902-946F-7D8C15462888	F.Jones	V0910996	Alice Johnson	EHR132312	Broken Leg	Orthopaedic
2	22A018D4-B803-4902-946F-7D8C15462888	F.Jones	V0910996	Alice Johnson	EHR400027	Diabetes	Endocrinology
3	22A018D4-B803-4902-946F-7D8C15462888	F.Jones	V0910996	Alice Johnson	EHR464642	T12 Crush Fracture	Orthopaedic
4	22A018D4-B803-4902-946F-7D8C15462888	F.Jones	R0098773	Bill Jones	EHR132367	Broken Leg	Orthopaedic
5	22A018D4-B803-4902-946F-7D8C15462888	F.Jones	R0098773	Bill Jones	EHR458668	Fractured Wrist	Orthopaedic
6	22A018D4-B803-4902-946F-7D8C15462888	F.Jones	R0098773	Bill Jones	EHR443533	Slipped Disk	Neurological

Screen 2: Accessible Records

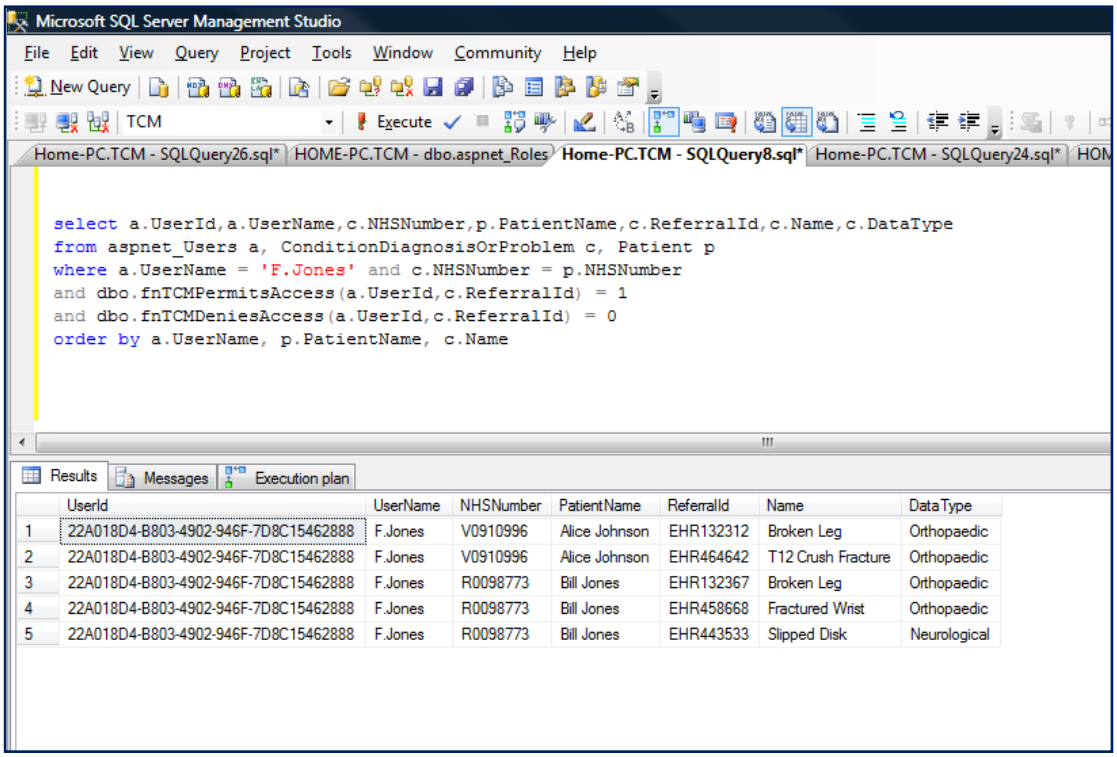
If the Deny CP

DCP = {<Role, GP>, <Security, Unsealed>, <DataType, Endocrinology>}

is added together with the inheritance relationship

<Role, HCP, GP>

then the results are:



Screen 3: Accessible Data Changed

The screen above shows that access to the diabetes record has been removed. If the PermitCP

> PCP = {<Role, GP>, <Security, Unsealed>, <DataType, Endocrinology>, <Location, Newlands>}

is added back in the output reverts to the original screen.

4.7 Summary

The TCM was re-examined in the light of the formal specification. The question of what makes the TCM unique and a contribution to access control was addressed. A model was specified in B that includes those essential components. To animate the model and as a step towards implementation the model was re-specified in Spec# using the B specification as a basis. Some model exploration was demonstrated, and some translation to the SQL that would be used in an implementation was given.

5 Case Study

This chapter is a case study that explores how TCM3 would be implemented as discussed in the previous chapter and given current health service thinking.

5.1 Introduction

This case study is based loosely on the original Fred and Alice scenario, but also incorporates new health care information thinking on „legitimate relationship" and „sealed and locked data". It demonstrates how these concepts can be easily implemented using TCM3.

For this case study, we envisage two local GP practices (Newlands, Linthorpe) and a local hospital (JCUH - James Cook University Hospital). All data is fictitious.

5.2 Logical Assertions

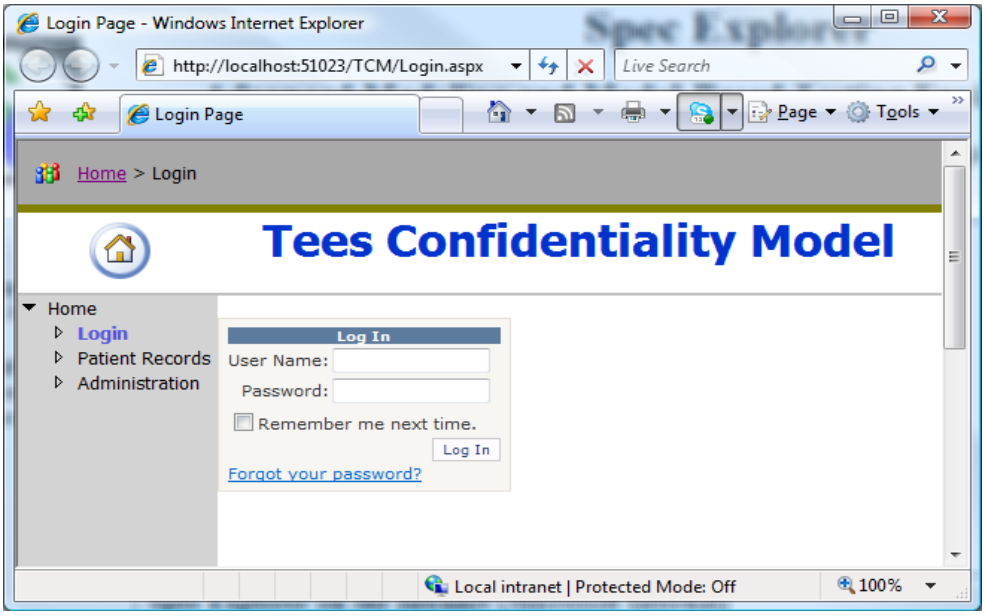
The model implements the following logical assertions.

1. A Health Care Professional can see a patient"s data if he has a legitimate relationship with the patient and the data is unsealed.
2. A Health Care Professional can see a patient"s data if he has a legitimate relationship with the patient and the data belongs to the same organisation and the same team/workgroup as the Health Care Professional.
3. A Health Care Professional can see a patient"s data if he has a legitimate relationship with the patient, the data is classified as sealed, the data does not belong to the same organisation and the same team/workgroup as the Health Care Professional and the access is audited.
4. A Health Care Professional cannot see a patient"s data if the data does not belong to the same organisation and the same team/workgroup as the Health Care Professional and the data is classified as locked.

5.3 Implementation

Various aspects of a TCM3 implementation are discussed below, together with screenshots of a working model.

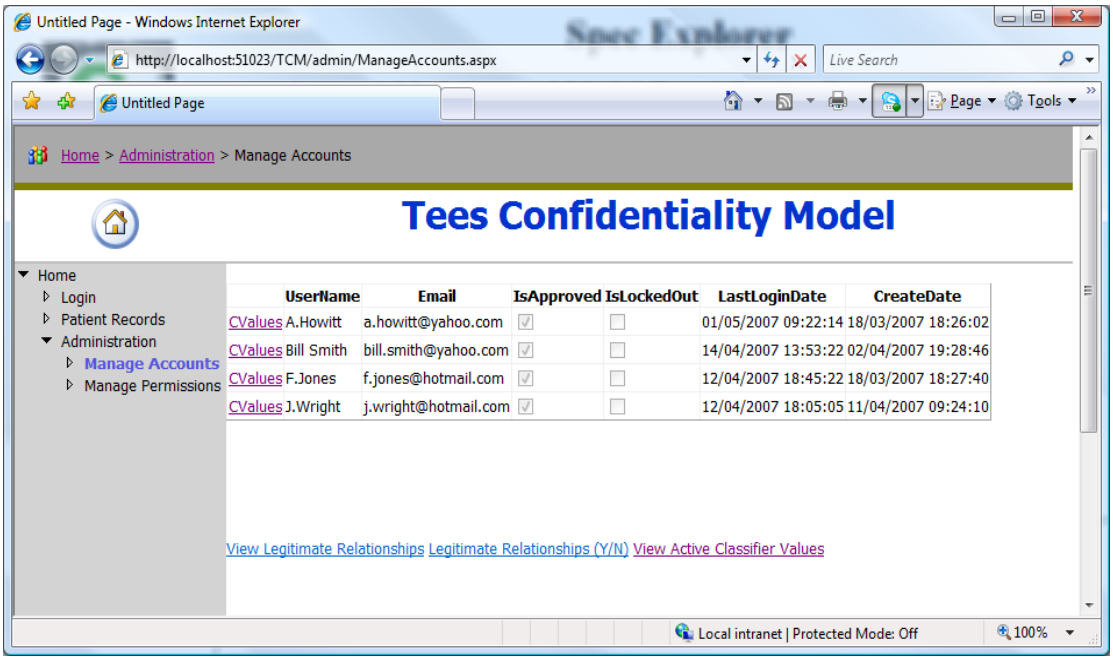
5.3.1 Login



Screen 4: Login

Login uses asp.net‟s membership and role providers. This has been modified to use classifiers and classifier values. Data is stored on SQL Server. Active Directory or anything using LDAP (Lightweight Directory Access Protocol) could be used. ADAM (Active Directory Application Mode) could also be used.

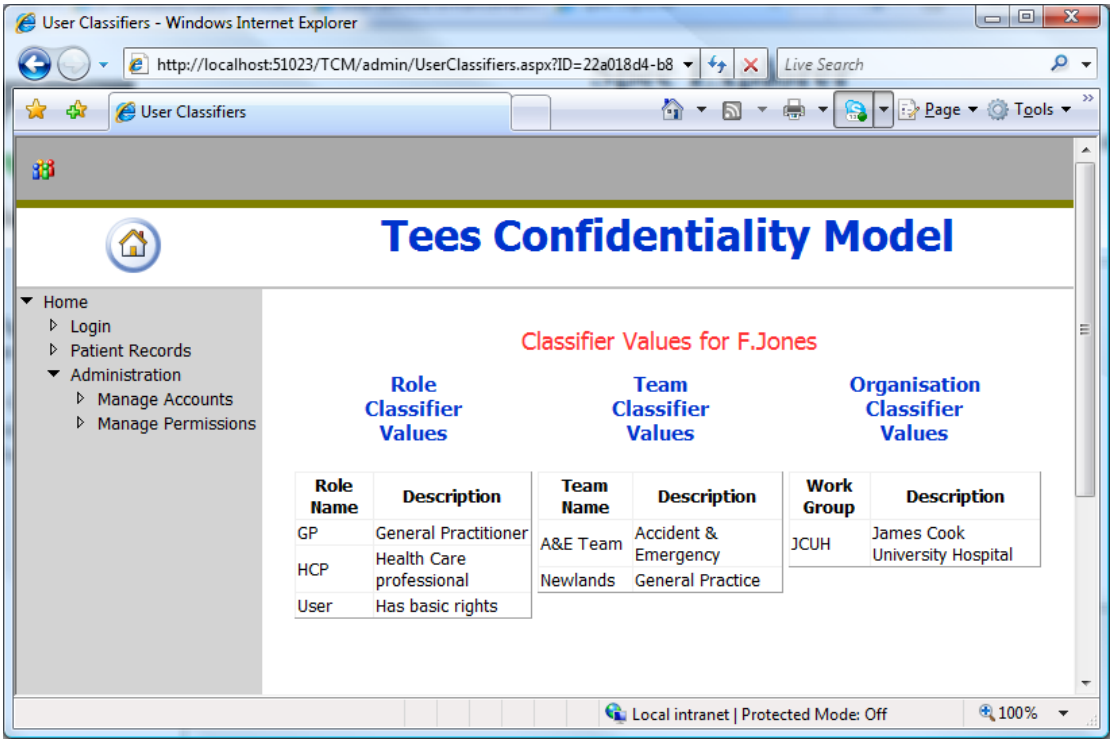
5.3.2 User Accounts



Screen 5: User Accounts

The above screen shows the user accounts, taken from asp.net‟s membership provider.

5.3.3 User Classifier Values



Screen 6: User Classifier Values

This above screen shows the classifier values active for a particular user.

5.3.4 Sealed and Locked Data

The application demonstrates the rules that apply to „normal", „sealed", and „sealed and locked" data. The rules for sealed data can be expressed as follows:

- A user in the same team/workgroup and organisation can see a patient"s sensitive (sealed) data. The user is to be made aware that it is sensitive data, but no warning or auditing is carried out.
- A user in a different team or organisation can also see sensitive data. However, a warning is given and access is audited.

If these were expressed using classifier values then they could be written as:

PCP1 = {<role, HCP>, <relativelocation, SOST>, <security, sealed>}

PCP2 = {<role, HCP>, <relativelocation, SODT>, <Audited, true>, <security, sealed>}

PCP3 = {<role, HCP>, <relativelocation, DO>, <Audited, true>, <security, sealed>}

HCP = „Health Care Professional"
SOST = „Same Organisation, Same Team/Workgroup"
SODT = „Same Organisation, Different Team/Workgroup"
DO = „Different Organisation"

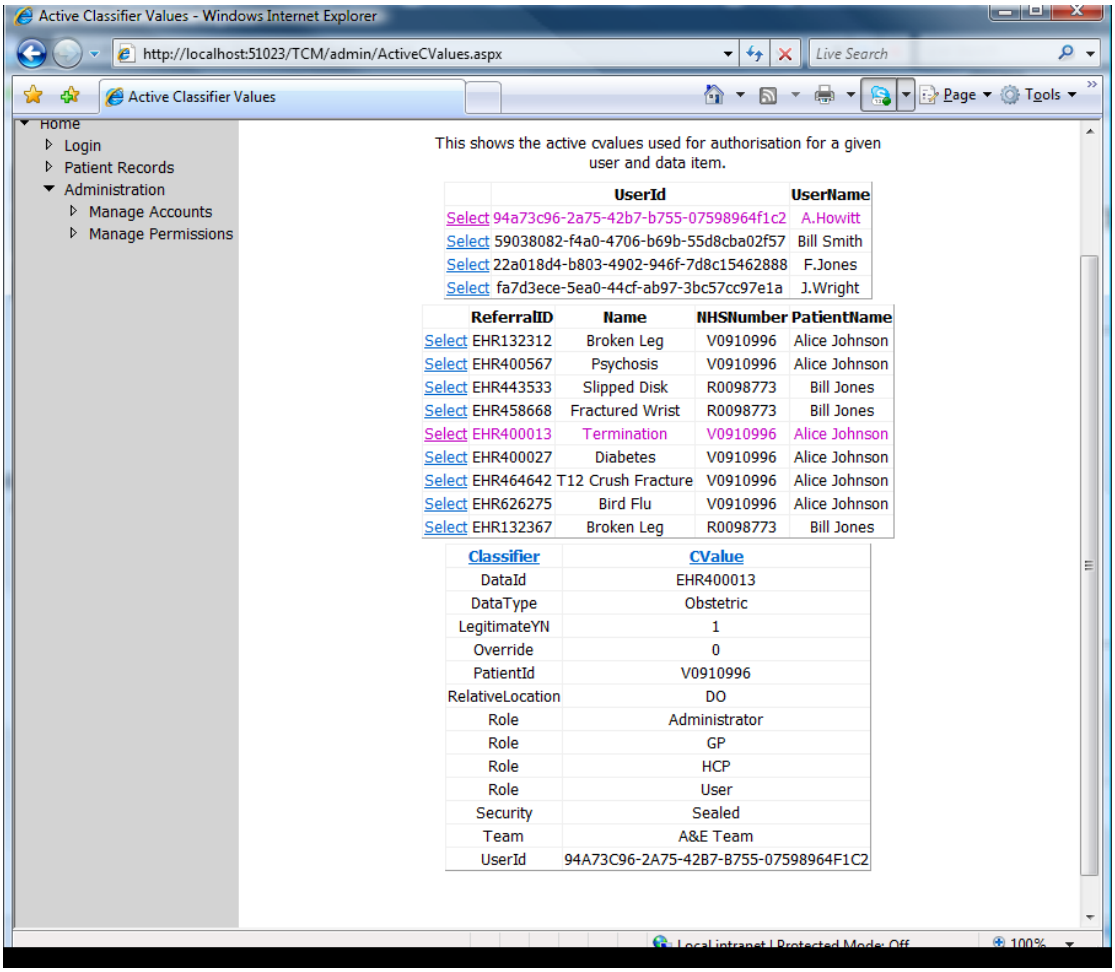
The rule for „sealed and locked" data is that it cannot be seen by anyone who is not in the same team and organisation. This could be expressed by two deny_cps.

DCP1 = {<role, HCP>, <relativelocation, SODT>, <op_type, view>, <security, locked>}
DCP2 = {<role, HCP>, <relativelocation, DO>, <op_type, view>, <security, locked>}

The application tests three users as follows:

Name	Password	Organisation	Location	Data Location
Bill Smith	billsmith	JCUH	Linthorpe	Newlands
F.Jones	ffones	JCUH	Newlands	Newlands
J.Wright	jwright	NTUH	A&E Team	Newlands

5.3.5 Active Classifier Values



Screen 7: Active Classifier Values

This screen shows the active classifier values for a particular user and data object. These are the classifier values that could be used for authorisation. The only requirement for a

value to be a „classifier value" is that it is persistent in the application. Note that LegitimateYN and RelativeLocation, which depend on both the user and the object, are present.

5.3.6 Permissions

Confidentiality Permissions can be stored as relational data in a database or as an XML file. The database tables are shown in below in Figure 9.

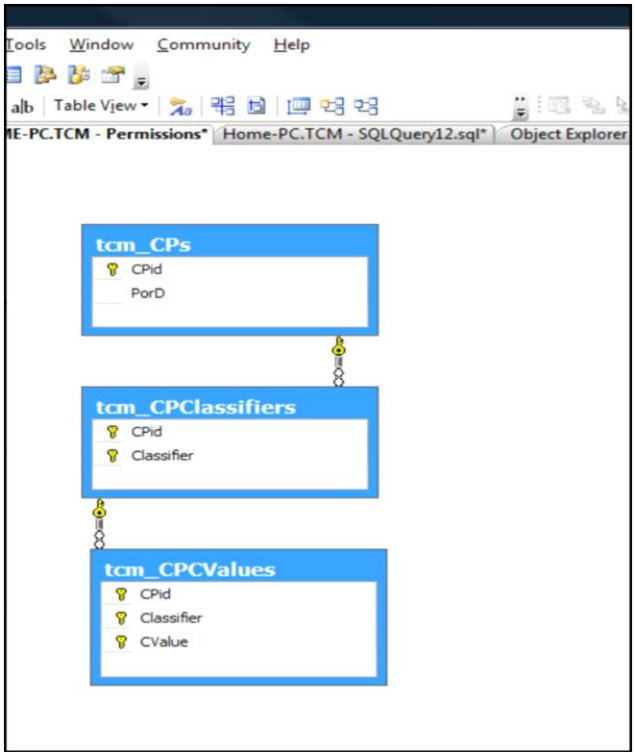


Figure 9: CP Tables

The XML below is for a CP that permits access in an audited/ override situation for sealed data in the same organisation, different team. Storing as an XML file would be in line with much current practice

```
<CP Cpid="8" Access="permit    ">
  <Classifier Name="Role">
    <CValue Value="HCP" />
  </Classifier>
  <Classifier Name="RelativeLocation">
    <CValue Value="SODT" />
  </Classifier>
  <Classifier Name="Security">
    <CValue Value="Sealed" />
  </Classifier>
  <Classifier Name="Override">
    <CValue Value="1" />
```

```
    </Classifier>
  </CP>
```

The CPs together with the inheritance relationship provide a complete permission description. Inheritance can be stored in XML as shown below.

```
  <Inheritance>
    <Classifier Name="Role">
      <Relationship Parent="HCP" Child="Registrar"/>
      <Relationship Parent="Registrar" Child="Consultant"/>
    </Classifier>
    <Classifier Name="Location">
      <Relationship Parent="TVAH" Child="JCUH"/>
      <Relationship Parent="JCUH" Child="JCUH A&E"/>
    </Classifier>
  </Inheritance>
```

Applying the inheritance to the CP above gives a new effective CP to be used for authorisation:

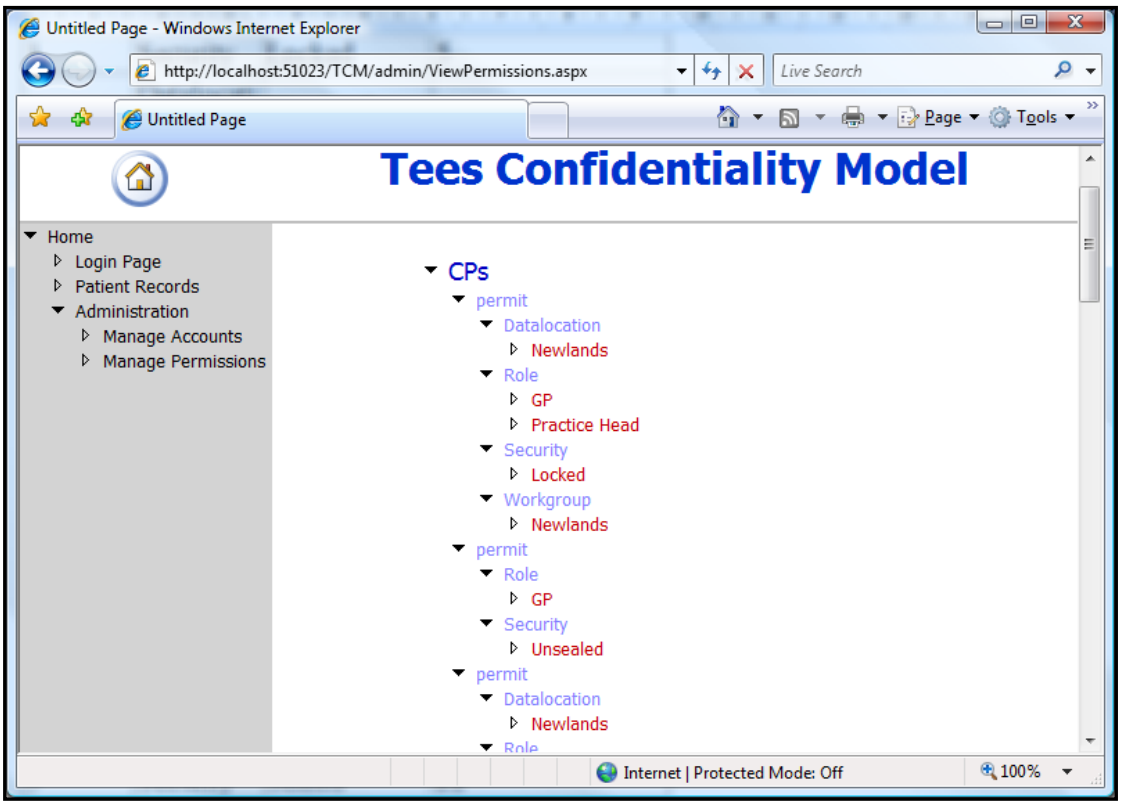
```
<CP Cpid="8" Access="permit    ">
    <Classifier Name="Role">
      <CValue Value="HCP" />
      <CValue Value="Registrar" />
      <CValue Value="Consultant" />
    </Classifier>
    <Classifier Name="RelativeLocation">
      <CValue Value="SODT" />
    </Classifier>
    <Classifier Name="Security">
      <CValue Value="Sealed" />
    </Classifier>
    <Classifier Name="Override">
      <CValue Value="1" />
    </Classifier>
  </CP>
```

Below is a stored procedure that takes advantage of functionality in SQL Server 2005 to manipulate XML files:

```
CREATE PROCEDURE [dbo].[spXMLCPs]
      -- Add the parameters for the stored procedure here
AS

BEGIN

      SELECT      CPid AS '@Cpid',
                  PorD AS '@Access',
                  (SELECT cpc.Classifier AS '@Name',
                  (SELECT CValue AS '@Value'
                  FROM TCM_CPCValues AS cpcv
                  WHERE cpc.CPid = cpcv.CPid
                  AND cpc.Classifier = cpcv.Classifier
                  FOR XML PATH ('CValue'), type)
                  FROM TCM_CPClassifiers AS cpc
                  WHERE cp.CPid = cpc.CPid
```

```
                    FOR XML PATH ('Classifier'), type)
                    FROM TCM_CPs AS cp
                    ORDER BY CPid
                    FOR XML PATH ('CP'), ROOT ('CPs')
END
```

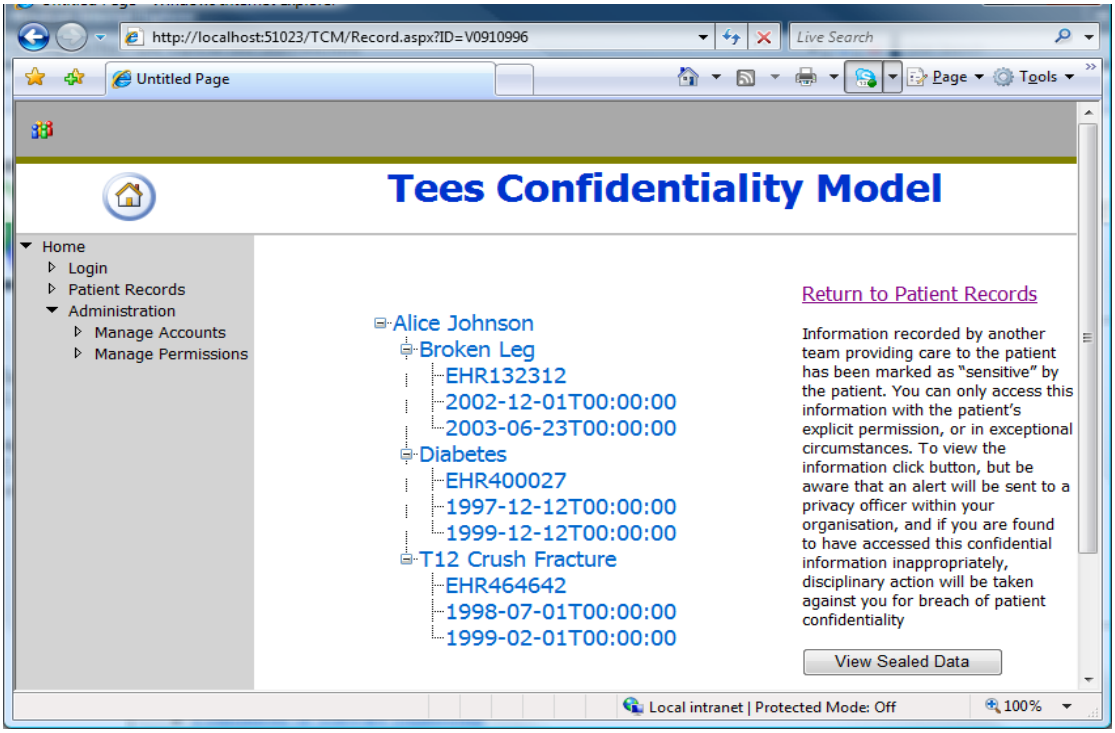
The screen print below shows some of the CPs used in the model. The display is produced using the above stored procedure.



Screen 8: CP Display

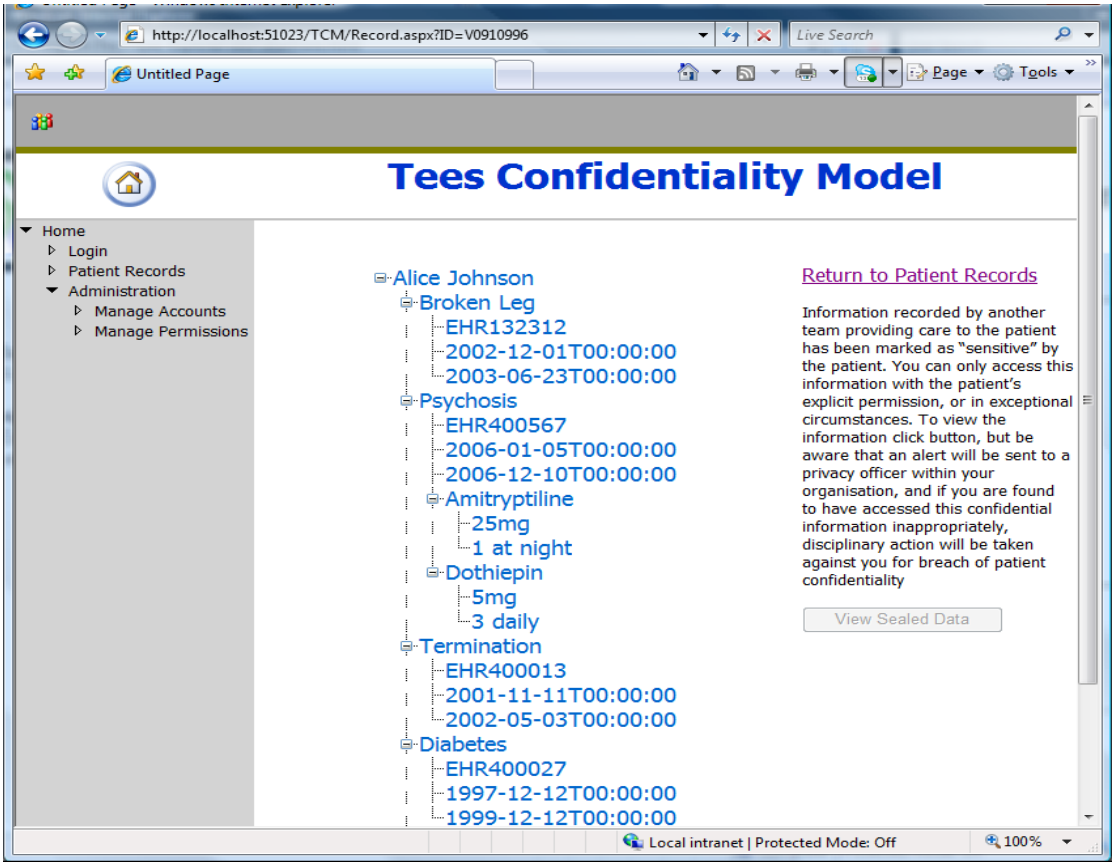
5.3.7 Patient Records

The screen below shows the patient details available to the current user through the *active classifier values* of both the user and the data objects.



Screen 9: Patient Records

The screen below demonstrates the different output when override is employed with the same user and data objects.



Screen 10: Patient Records with Override

5.4 Comparison of TCM3 with RBAC

RBAC greatly simplified access control through the assignment of roles to users, and the assignment of permissions to roles, thus modelling the real world and allowing for change of personnel within an organisation. Any new employee has only to be assigned to the appropriate role(s) for them to receive the correct permissions if the role permissions are set correctly. Permissions design for roles in RBAC is thus a large area of study and is part of the „administration" that goes together with RBAC.

However faced with legitimate access assertions such as

- GP Fred is allowed access to orthopaedic patient records at James Cook University Hospital if he is employed as an out-patients" session-doctor at the hospital, or
- A trainee pharmacist can read but not write prescriptions only in the dispensing clinic, or
- A research paper cannot be reviewed by someone belonging to the same department as the author, or who has previously co-authored a paper with the author, or
- An assistant engineer has all the permissions of an engineer if he is acting in a delegated role.

It is difficult to see how RBAC can be applied without a lot of tinkering with the basic concepts, or with very elaborate permissions design. In general, this is the way that progress has gone – keeping the basic idea of RBAC but with added extras.

It is in this area we see the benefits of the TCM in general and in particular TCM3. The TCM takes access control back to first principles. This had already been done intuitively to a large extent by Dr. Longstaff. The application of formal methods enables this „back to first principles" to be rigorously applied. Any logical assertion, such as those above, can be converted to a Confidentiality Permission using persistent classifier values. Moreover, in a database such as SQL Server all classifier values can be made persistent by supplying default values.

Another name for the TCM could be CBAC (Classifier Based Access Control) i.e. a generalisation of RBAC. A moment"s study of the access assertions above enable the classifiers to be picked out - Role, Identity, Data Type, Patient Identity, Location,

Employment Status etc. etc. This complexity is endemic to the TCM whilst the mainstream has continued to be RBAC centred with add-ons.

5.5 Comparison of TCM3 with TCM1, TCM2.

It is not claimed that TCM3 is an improved development of TCM1 and TCM2. What is claimed is that it is an improvement on RBAC and contains the essential ideas of the TCM in their most elemental form. Features of TCM1 and TCM2 can be added back into TCM3 but it is the contention that TCM3 can cope with most (if not all) authorisation situations.

The day-to-day demands on a systems administrator can be exacting. When applying a new permission or prohibition the system‟s administrator wants to know clearly the effect it is going to have. They would still want to know this even when working with different CPTs as in the more general models of TCM1 and TCM2. E.g. an elemental question such as “Can Fred Smith with a GP role at Newlands Medical Practice access Alice Jone‟s obstetric data?”. No doubt, there can be some graphical display that interprets the different layers of TCM1 and TCM2 to this effect, but the point here is that TCM3 implements this directly, and if it can be implemented directly why add extra layers of complication.

In TCM2 Confidentiality Permission Types act as a constraint on the CPs that can be created. Additionally an ordering is placed on the CPTs that changes how the CPs are processed. The question arises as to whether that this unnecessarily complicates things for the system administrator. If it does, then in practice, what happens then is that greater permissions than necessary are allocated in response to petitions from users that they do not have the permissions they need. This defeats the point of a having a complicated system in the first place.

In TCM3, the administrator has an instant response from the way the TCM3 is set up e.g. you do not have permission because although James Cook Consultants normally have read access to administrative data, they do not have access to financial data in the Personnel department.

One of the criticisms levelled at TCM2 in one of the paper submissions was as follows:

“Why should {Role, EHR_object_id} precede {Role, EHR_obj_type}? How do you define "more complex"?”

When approaching the model from a mathematical viewpoint the same questions arise. The answer from the viewpoint of the TCM2 is that the classifier ordering is determined by

the application designer, and that there is no special reason why one classifier should be more important than the other is. The classifier ordering determines the CPT ordering. The actual rules for „more complex" are given in the TCM2 specification, but basically, „more classifiers" in the CPT means „more complex".

In TCM3, the requirement that there be an ordering on classifiers or classifier types has been removed, although this could be added back in to meet the demands of a particular application.

The essential thinking behind the phrase „more complex" has been kept in that more complex CPs refine less complex CPs, although the idea that any „more complex" CP has precedence over a less complex CP irrespective of whether it refines that CP has been discarded.

TCM3 also removes the idea of overrides. Instead, it treats particular override situations as classifiers and classifier values e.g. <condition, orange> for a security situation. Perhaps more controversially TCM3 looks at authentication as a classifier with values e.g. <authenticated, yes>. A session is then just defined as that period during which a user is continuously authenticated.

5.6 Summary

A TCM3 model implementation of a modified „Alice" scenario has been produced using current health service thinking on sealed and locked data. The implementation of Confidentiality Permissions stored as tables in a relational database or in XML was explored. The implementation enabled some comparisons to be made between TCM3 and RBAC, and between TCM3 and TCM1 and TCM2.

6 TCM Classes using LinQ

6.1 Introduction

This section shows the creation of a TCM class which can then be used as an ObjectDataSource in the development of an application. The methods of this class are written using the relatively new LinQ described below, although they could be written using the more usual ADO.NET procedure calls.

6.2 LinQ

LINQ Project is a set of extensions to the .NET Framework that encompass language-integrated query, set, and transform operations. It extends C# and Visual Basic with native language syntax for queries and provides class libraries to take advantage of these capabilities. It is an attempt to bridge the divide between programming languages such as C# and database languages such as TRANSACT SQL. It enables calls to any data source, including XML, using programming language syntax.

6.3 The Cfier Class

This class is used in TCM1, TCM2. It is not needed in TCM3 because TCM3 works primarily with a cvalue i.e. a classifier, value combination. TCM1, TCM2 have Confidentiality Permission Types which are sets of classifiers.

```
public class cfier
{
    public string CFIER { get; set; }


    public override bool Equals(object obj)
    {
        if (!(obj is cfier))
            return false;

        else
        {
            cfier cf = (cfier)obj;
            return (cf.CFIER.Trim() == this.CFIER.Trim());
        }
    }

    public override int GetHashCode()
    {
        return string.Format("{0}", this.CFIER.Trim()).GetHashCode();
    }
}
```

6.4 The CValue Class

The CValue class is used by TCM1, TCM2, and TCM3.

```
public class cvalue
{
    public string CFIER { get; set; }
    public string VAL { get; set; }

    public override bool Equals(object obj)
    {
        if (!(obj is cvalue))
            return false;

        else
        {
            cvalue cv = (cvalue)obj;
            return (cv.CFIER.Trim() == this.CFIER.Trim() &&
                cv.VAL.Trim() == this.VAL.Trim());
        }
    }

    public override int GetHashCode()
    {
        return string.Format("{0}|{1}", this.CFIER.Trim(),
this.VAL.Trim()).GetHashCode();
    }
}
```

6.5 The TCM Class

Ninety percent of the methods for TCM1, TCM2, and TCM3 are in common. The following specifies a TCM class of those common methods which can used as an authorisation source for any application,

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Linq.Expressions;
using System.Collections;
using System.Collections.Generic;


public class TCM
{
    public static void AddUser(string FirstName,
                               string SecondName,
                               string LogonName,
                               string Password)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = new USER();
        user.FirstName = FirstName;
        user.SecondName = SecondName;
        user.LogonName = LogonName;
        user.Password = Password;
        db.USERs.Add(user);
        db.SubmitChanges();
    }

    public static void DeleteUser(Int32 UserId)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.UserId == UserId);
        db.USERs.Remove(user);
        db.SubmitChanges();
    }
```

```
    public static void UpdateUser(Int32 UserId,
                                  string FirstName,
                                  string SecondName,
                                  string LogonName,
                                  string Password)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.UserId == UserId);
        user.FirstName = FirstName;
        user.SecondName = SecondName;
        user.LogonName = LogonName;
        user.Password = Password;
        db.SubmitChanges();
    }

    public static object ViewUsers()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from u in db.USERs select u;
        return query;
    }


    public static object ViewUser(Int32 UserId)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from u in db.USERs
                    where u.UserId == UserId
                    select u;
        return query;
    }

    public static USER FindUser(string LogonName)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.LogonName == LogonName);
        return user;
    }

    public static void AssignUser(Int32 UserId, string Cfier, string
Value)
    {
        TCMDataContext db = new TCMDataContext();
        UA ua = new UA();
        ua.UserId = UserId;
        ua.Cfier = Cfier;
        ua.Value = Value;
        db.UAs.Add(ua);
        db.SubmitChanges();
    }

    public static void DeassignUser(Int32 UserId, string Cfier, string
Value)
    {
        TCMDataContext db = new TCMDataContext();
        UA ua = db.UAs.Single(u => u.UserId == UserId &&
                                   u.Cfier == Cfier &&
                                   u.Value == Value);
        db.UAs.Remove(ua);
        db.SubmitChanges();
    }

    public static object ViewUAs()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from ua in db.UAs select ua;
        return query;
    }
```

```
    public static object ViewUAs(Int32 UserId)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from ua in db.UAs
                    where ua.UserId == UserId
                    select ua;
        return query;
    }

    public static object ViewOBAs()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from oba in db.OBAs select oba;
        return query;
    }

    public static object ViewOBAs(Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from oba in db.OBAs
        where oba.RecordId == RecordId select oba;
        return query;
    }

    public static object ViewUOBAs()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from uoba in db.UOBAs select uoba;
        return query;
    }

    public static object ViewUOBAs(Int32 UserId, Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from uoba in db.UOBAs
                    where uoba.UserId == UserId &&
                    uoba.RecordId == RecordId select uoba;
        return query;
    }

    public static void AddPermitCPElement(Int32 PcpId, string Cfier,
string Value)
    {
        TCMDataContext db = new TCMDataContext();
        PERMITCP pcp = new PERMITCP();
        pcp.PcpId = PcpId;
        pcp.Cfier = Cfier;
        pcp.Value = Value;
        db.PERMITCPs.Add(pcp);
        db.SubmitChanges();
    }

    public static void RemovePermitCPElement(Int32 PcpId, string Cfier,
string Value)
    {
        TCMDataContext db = new TCMDataContext();
        PERMITCP pcp = db.PERMITCPs.Single(p => p.PcpId == PcpId &&
                                p.Cfier == Cfier &&
                                p.Value == Value);
        db.PERMITCPs.Remove(pcp);
        db.SubmitChanges();
    }

    public static void RemovePermitCP(Int32 PcpId)
    {
        TCMDataContext db = new TCMDataContext();
        PERMITCP pcp = db.PERMITCPs.Single(p => p.PcpId == PcpId);
        db.PERMITCPs.Remove(pcp);
        db.SubmitChanges();
```

```
    }

    public static object ViewPermitCPs()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from pcp in db.PERMITCPs
                    orderby pcp.PcpId, pcp.Cfier, pcp.Value
                    select pcp;
        return query;
    }


    public static object ViewPermitCP(Int32 PcpId)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from pcp in db.PERMITCPs
                    where pcp.PcpId == PcpId
                    select pcp;
        return query;
    }

    public static void AddDenyCPElement(Int32 DcpId, string Cfier,
string Value)
    {
        TCMDataContext db = new TCMDataContext();
        DENYCP dcp = new DENYCP();
        dcp.DcpId = DcpId;
        dcp.Cfier = Cfier;
        dcp.Value = Value;
        db.DENYCPs.Add(dcp);
        db.SubmitChanges();
    }

    public static void RemoveDenyCPElement(Int32 DcpId, string Cfier,
string Value)
    {
        TCMDataContext db = new TCMDataContext();
        DENYCP dcp = db.DENYCPs.Single(p => p.DcpId == DcpId &&
                                  p.Cfier == Cfier &&
                                  p.Value == Value);
        db.DENYCPs.Remove(dcp);
        db.SubmitChanges();
    }

    public static void RemoveDenyCP(Int32 DcpId)
    {
        TCMDataContext db = new TCMDataContext();
        DENYCP dcp = db.DENYCPs.Single(p => p.DcpId == DcpId);
        db.DENYCPs.Remove(dcp);
        db.SubmitChanges();
    }

    public static object ViewDenyCPs()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from dcp in db.DENYCPs
                    orderby dcp.DcpId, dcp.Cfier, dcp.Value
                    select dcp;
        return query;
    }


    public static object ViewDenyCP(Int32 DcpId)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from dcp in db.DENYCPs
                    where dcp.DcpId == DcpId
                    select dcp;
        return query;
```

```
    }

    public static void AddInheritance(string Cfier, string Pval, string
Cval)
    {
        TCMDataContext db = new TCMDataContext();
        CPC cpc = new CPC();
        cpc.Cfier = Cfier;
        cpc.Pval = Pval;
        cpc.Cval = Cval;
        db.CPCs.Add(cpc);
        db.SubmitChanges();
    }

    public static void DeleteInheritance(string Cfier, string Pval,
string Cval)
    {
        TCMDataContext db = new TCMDataContext();
        CPC cpc = db.CPCs.Single(i => i.Cfier == Cfier &&
                                 i.Pval == Pval &&
                                 i.Cval == Cval);
        db.CPCs.Remove(cpc);
        db.SubmitChanges();
    }

    public static object ViewInheritance()
    {
        TCMDataContext db = new TCMDataContext();
        var query = from cpc in db.CPCs
                    orderby cpc.Cfier, cpc.Pval, cpc.Cval
                    select cpc;
        return query;
    }


    public static IEnumerable<cvalue> AssignedCValues(Int32 UserId,
Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();


       IEnumerable<cvalue> ucvs = db.UAs.Where(ua => ua.UserId ==
UserId).Select(ua =>
            new cvalue {CFIER = ua.Cfier,VAL = ua.Value });

        IEnumerable<cvalue> obcvs = db.OBAs.Where(oba => oba.RecordId ==
       RecordId).Select(oba =>
            new cvalue { CFIER = oba.Cfier, VAL = oba.Value });

        IEnumerable<cvalue> uobcvs = db.UOBAs.Where(uoba => uoba.UserId
== UserId &&
                    uoba.RecordId == RecordId).Select(uoba =>
            new cvalue { CFIER = uoba.Cfier, VAL = uoba.Value });

        IEnumerable<cvalue> query = ucvs.Union(obcvs).Union(uobcvs);

        return query;

    }


    public static bool PermitAccess(Int32 PcpId, Int32 UserId, Int32
RecordId)
    {
        var cfiersofcommoncvalues = AssignedCValues(UserId,
RecordId).Intersect(
            PermitCpCValues(PcpId)).Select(a => a.CFIER.Trim());

        bool permit =
```

```
             (PermitCpCValues(PcpId).Select(p=>p.CFIER.Trim())
              .Except(cfiersofcommoncvalues)).Count() == 0;

        return permit;

    }

    public static bool DenyAccess(Int32 DcpId, Int32 UserId, Int32
RecordId)
    {
        var cfiersofcommoncvalues = AssignedCValues(UserId,
RecordId).Intersect(
            DenyCpCValues(DcpId)).Select(a => a.CFIER.Trim());

        bool deny =
             (DenyCpCValues(DcpId).Select(d => d.CFIER.Trim()).
               Except(cfiersofcommoncvalues)).Count() == 0;

        return deny;
    }


    //helper methods
    public static bool PermitRefines(Int32 DcpId, Int32 PcpId)
    {
        TCMDataContext db = new TCMDataContext();

        IEnumerable<string> domDenyCp = from d in db.DENYCPs
                                        where (d.DcpId == DcpId)
                                        select d.Cfier.Trim();

        IEnumerable<string> domPermitCp = from p in db.PERMITCPs
                                          where (p.PcpId == PcpId)
                                          select p.Cfier.Trim();


        if (domDenyCp.Except(domPermitCp).Count() == 0)
        {
            IEnumerable<cvalue> pdescs =
                RestrictedPermitCpCValues(PcpId, DcpId);
            IEnumerable<cvalue> ddescs =
                DenyCpCValues(DcpId);

            int PexD = domPermitCp.Except(domDenyCp).Count();
            int DexP = ddescs.Except(pdescs).Count();

            if ((PexD > 0) && (DexP >= 0))
            {
                return true;
            }
            else
            {

                if ((PexD == 0) && (DexP > 0))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }
        else
        {
            return false;
        }

    }
```

```
    public static bool DenyRefines(Int32 PcpId, Int32 DcpId)
    {
        TCMDataContext db = new TCMDataContext();

        IEnumerable<string> domPermitCp = from p in db.PERMITCPs
                                          where (p.PcpId == PcpId)
                                          select p.Cfier.Trim();

        IEnumerable<string> domDenyCp = from d in db.DENYCPs
                                        where (d.DcpId == DcpId)
                                        select d.Cfier.Trim();



        if (domPermitCp.Except(domDenyCp).Count() == 0)
        {
            IEnumerable<cvalue> ddescs =
                RestrictedDenyCpCValues(DcpId, PcpId);
            IEnumerable<cvalue> pdescs =
                PermitCpCValues(PcpId);

            int DexP = domDenyCp.Except(domPermitCp).Count();
            int PexD = pdescs.Except(ddescs).Count();


            if ((DexP > 0) && (PexD >= 0))
            {
                return true;
            }
            else
            {
                if ((DexP == 0) && (PexD > 0))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }
        else
        {
            return false;
        }

    }


    public static IEnumerable<string> Dom(IEnumerable<cvalue> cvs)
    {
        TCMDataContext db = new TCMDataContext();
        var query = from cv in cvs select cv.CFIER.Trim(); ;
        return query;
    }


    public static IEnumerable<cvalue> PermitCpCValues(Int32 PcpId)
    {
        TCMDataContext db = new TCMDataContext();

        IEnumerable<cvalue> query = from pcp in db.spPERMITCPS(PcpId)
                    select new cvalue { CFIER = pcp.Cfier.Trim(), VAL =
pcp.Value.Trim() };
        return query;
    }
```

```
    public static IEnumerable<cvalue> DenyCpCValues(Int32 DcpId)
    {
        TCMDataContext db = new TCMDataContext();

        IEnumerable<cvalue> query = db.spDENYCPS(DcpId).Select(dcp =>
            new cvalue { CFIER = dcp.Cfier.Trim(), VAL =
dcp.Value.Trim()});

        return query;
    }

    public static IEnumerable<cvalue> RestrictedPermitCpCValues(Int32
PcpId, Int32 DcpId)
    {
        TCMDataContext db = new TCMDataContext();
        IEnumerable<cvalue> pdescs
                = from pcp in db.spPERMITCPS(PcpId)
                  from d in db.DENYCPs
                  where (d.DcpId == DcpId) &&
                  d.Cfier.Trim() == pcp.Cfier.Trim()
                  select new cvalue { CFIER = pcp.Cfier.Trim(), VAL =
pcp.Value.Trim() };

        return pdescs;
    }

    public static IEnumerable<cvalue> RestrictedDenyCpCValues(Int32
DcpId, Int32 PcpId)
    {
        TCMDataContext db = new TCMDataContext();
        IEnumerable<cvalue> ddescs
                = from dcp in db.spDENYCPS(DcpId)
                  from p in db.PERMITCPs
                  where (p.PcpId == PcpId) &&
                  p.Cfier.Trim() == dcp.Cfier.Trim()
                  select new cvalue { CFIER = dcp.Cfier.Trim(), VAL =
dcp.Value.Trim()};

        return ddescs;
    }


}
```

6.6 TCM2 Class

A TCM2 class is specified which derives from the TCM class.

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Linq.Expressions;
using System.Collections;
using System.Collections.Generic;

public class TCM2:TCM
{
    public static bool CptToPermitCpMatch(Int32 CptId, Int32 PcpId)
    {
        TCMDataContext db = new TCMDataContext();

        bool result = false;
```

```
        IEnumerable<cfier> PcpCfiers = db.PERMITCPs.Where(p => p.PcpId
== PcpId).Select(p => new cfier{ CFIER = p.Cfier.Trim() }).Distinct();
        IEnumerable<cfier> CptCfiers = db.CPTs.Where(c => c.CptId ==
CptId).Select(c => new cfier { CFIER = c.Cfier.Trim() }).Distinct();

        if (PcpCfiers.Except(CptCfiers).Count() == 0 &&
CptCfiers.Except(PcpCfiers).Count() == 0)
        {
            result = true;
        }

        return result;
    }

    public static bool CptToDenyCpMatch(Int32 CptId, Int32 DcpId)
    {
        TCMDataContext db = new TCMDataContext();

        bool result = false;

        IEnumerable<cfier> DcpCfiers = db.DENYCPs.Where(d => d.DcpId ==
DcpId).Select(d => new cfier { CFIER = d.Cfier.Trim() }).Distinct();

        IEnumerable<cfier> CptCfiers = db.CPTs.Where(c => c.CptId ==
CptId).Select(c => new cfier { CFIER = c.Cfier.Trim() }).Distinct();

        if (DcpCfiers.Except(CptCfiers).Count() == 0 &&
CptCfiers.Except(DcpCfiers).Count() == 0)
        {
            result = true;
        }

        return result;
    }

    public static bool CptPermitAccess(Int32 CptId, Int32 UserId, Int32
RecordId)
    {
        TCMDataContext db = new TCMDataContext();
        bool result = false;
            foreach (PERMITCP pcp in db.PERMITCPs)
            {
                if (PermitAccess(pcp.PcpId, UserId, RecordId)&&
                    CptToPermitCpMatch(CptId,pcp.PcpId))
                {
                    result = true;
                }
                break;
            }
        return result;
    }

    public static bool CptDenyAccess(Int32 CptId, Int32 UserId, Int32
RecordId)
    {
        TCMDataContext db = new TCMDataContext();
        bool result = false;
        foreach (DENYCP dcp in db.DENYCPs)
        {
            if (DenyAccess(dcp.DcpId, UserId, RecordId) &&
                CptToDenyCpMatch(CptId, dcp.DcpId))
            {
                result = true;
            }
            break;
        }
        return result;
    }
```

```
    public static bool TCM2PermitAccess(Int32 UserId, Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();
        bool result = false;

        IQueryable<CPT> cpts = from c in db.CPTs
                               orderby c.CptId ascending
                               select c;

        foreach (CPT cpt in cpts)
        {

            if (CptDenyAccess(cpt.CptId, UserId, RecordId))
            {
                break;
            }

            if (CptPermitAccess(cpt.CptId, UserId, RecordId))
            {
                result = true;
                break;
            }
        }
        return result;
     }

    //This includes an override level.
    public static bool TCM2PermitAccess(Int32 UserId, Int32 RecordId,
Int16 OverrideLevel)
    {
        TCMDataContext db = new TCMDataContext();
        bool result = false;

        IQueryable<CPT> cpts = from c in db.CPTs
                               orderby c.CptId ascending
                               select c;

        foreach (CPT cpt in cpts)
        {

            if (cpt.CptId > OverrideLevel && (CptDenyAccess(cpt.CptId,
UserId, RecordId)))
            {
                break;
            }

            if (CptPermitAccess(cpt.CptId, UserId, RecordId))
            {
                result = true;
                break;
            }
        }
        return result;
    }

    public static object Permissions(Int32 UserId)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM2PermitAccess(UserId, r.RecordId))
            {
                perms.Add(r);
            }
```

```
        }
        return perms;
    }

    public static object Permissions(Int32 UserId, Int16 OverrideLevel)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM2PermitAccess(UserId, r.RecordId, OverrideLevel))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object Permissions(String LogonName)
    {
        TCMDataContext db = new TCMDataContext();

        USER user = db.USERs.Single(u => u.LogonName == LogonName);
        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM2PermitAccess(user.UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object Permissions(String LogonName, Int16
OverrideLevel)
    {
        TCMDataContext db = new TCMDataContext();

        USER user = db.USERs.Single(u => u.LogonName == LogonName);
        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM2PermitAccess(user.UserId, r.RecordId,
OverrideLevel))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object Permissions(Int32 UserId, Int32 PatientId)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
        {
            if (TCM2PermitAccess(UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
```

```
    }

    public static object Permissions(Int32 UserId, Int32 PatientId,
Int16 OverrideLevel)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
        {
            if (TCM2PermitAccess(UserId, r.RecordId, OverrideLevel))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object Permissions(String LogonName, Int32 PatientId)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.LogonName == LogonName);

        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
        {
            if (TCM2PermitAccess(user.UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }

        return perms;

    }

    public static object Permissions(String LogonName, Int32 PatientId,
Int16 OverrideLevel)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.LogonName == LogonName);

        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
        {
            if (TCM2PermitAccess(user.UserId, r.RecordId,
OverrideLevel))
            {
                perms.Add(r);
            }
        }

        return perms;

    }

    public static bool TCM2GlobalAccess(Int32 UserId, Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();

        bool result = false;
        foreach (PERMITCP pcp in db.PERMITCPs)
        {
            if (PermitAccess(pcp.PcpId, UserId, RecordId))
            {
```

```
                result = true;
                break;
            }
        }

        return result;
    }

    public static object GlobalPermissions(Int32 UserId)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM2GlobalAccess(UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
    }



    public static object GlobalPermissions(String LogonName)
    {
        TCMDataContext db = new TCMDataContext();

        USER user = db.USERs.Single(u => u.LogonName == LogonName);
        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM2GlobalAccess(user.UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object GlobalPermissions(Int32 UserId, Int32
PatientId)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
        {
            if (TCM2GlobalAccess(UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object GlobalPermissions(String LogonName, Int32
PatientId)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.LogonName == LogonName);

        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
```

```
        {
            if (TCM2GlobalAccess(user.UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }

        return perms;

    }


}
```

6.7 TCM3 Class

A TCM3 class is specified which derives from the TCM class. TCM2 and TCM3 differ in that TCM2 uses CPTs and the permissions are derived in a different way. There are fewer methods in the TCM class, because override is regarded as just another classifier, and there is no necessity to match to Confidentiality Permission Types.

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Linq.Expressions;
using System.Collections;
using System.Collections.Generic;
using System.Xml.Linq;

public class TCM3:TCM
{
    public static bool PermitAccess(Int32 UserId, Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();

        bool result = false;
        foreach (PERMITCP pcp in db.PERMITCPs)
        {
            if (PermitAccess(pcp.PcpId, UserId, RecordId))
            {
                result = true;
                foreach (DENYCP dcp in db.DENYCPs)
                {
                    if (DenyAccess(dcp.DcpId, UserId, RecordId) &&
                        DenyRefines(pcp.PcpId, dcp.DcpId))
                    {
                        result = false; ;
                        break;
                    }
                }
                break;
            }
        }

        return result;
    }

    public static bool DenyAccess(Int32 UserId, Int32 RecordId)
    {
        TCMDataContext db = new TCMDataContext();
        bool result = false;
        foreach (DENYCP dcp in db.DENYCPs)
```

```
        {
            if (DenyAccess(dcp.DcpId, UserId, RecordId))
            {
                result = true;
                foreach (PERMITCP pcp in db.PERMITCPs)
                {
                    if (PermitAccess(pcp.PcpId, UserId, RecordId) &&
                        PermitRefines(dcp.DcpId, pcp.PcpId))
                    {
                        result = false;
                        break;
                    }
                }
                break;
            }
        }

        return result;
    }



    public static bool TCM3PermitAccess(Int32 UserId, Int32 RecordId)
    {


        if (DenyAccess(UserId, RecordId))
        {
            return false;
        }
        else
        {
            if (PermitAccess(UserId, RecordId))
            {
                return true;
            }
        }
        return false;
    }

    public static object Permissions(Int32 UserId)
    {
        TCMDataContext db = new TCMDataContext();


        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM3PermitAccess(UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object Permissions(String LogonName)
    {
        TCMDataContext db = new TCMDataContext();

        USER user = db.USERs.Single(u => u.LogonName == LogonName);
        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs)
        {
            if (TCM3PermitAccess(user.UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
```

```
        return perms;
    }

    public static object Permissions(Int32 UserId, Int32 PatientId)
    {
        TCMDataContext db = new TCMDataContext();

        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
PatientId))
        {
            if (TCM3PermitAccess(UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }
        return perms;
    }

    public static object Permissions(String LogonName, Int32 PatientId)
    {
        TCMDataContext db = new TCMDataContext();
        USER user = db.USERs.Single(u => u.LogonName == LogonName);

        List<RECORD> perms = new List<RECORD>();
        foreach (RECORD r in db.RECORDs.Where(rec => rec.PatientId ==
                                                        PatientId))
        {
            if (TCM3PermitAccess(user.UserId, r.RecordId))
            {
                perms.Add(r);
            }
        }

        return perms;

    }
}
```

6.8 Using the TCM class as a source.

The piece of HTML below is part of the source code for a webpage that displays, adds, and deletes users:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
            DeleteMethod="DeleteUser" InsertMethod="AddUser"
SelectMethod="ViewUsers"
            TypeName="TCM" UpdateMethod="UpdateUser">
```

The object data source is used the source for a GridView and a Details View. It can be seen that it is of type TCM and is using the methods of the TCM class. The TCM class can be similarly used to show and use all the features of the TCM.

6.9 Active Classifiers Values

Active classifiers values are those classifier values present in a system that the confidentiality permissions use to permit or deny access. The method that returns the set of active classifier values for a particular user, operation, and object is unique for a TCM

application instance. This method is written as an interface and requires detailing for each instance.

The source of user classifier values is usually user assignment. Operation and object classifiers may be properties or just the operation and object identifiers. Some classifiers e.g. those that depend on both user and object may need to be derived or obtained from a dedicated server. Examples of those would be “relative location” and “legitimate relationship”.

LinQ is ideal for collecting the set of all active classifier values, because of its ability to combine data from different data sources using the same syntax.

6.10 Visual Studio Website

The website is to demonstrate the use of the TCM class and the use of LinQ to SQL. In order to demonstrate these we revisit Alice‟s scenario and add some additional requirements

6.10.1 Alice’s Scenario Revisited

To look at Alice‟s scenario in the light of current health service thinking then Alice has the right to mark her health data as „sealed‟ or „sealed and locked‟. Let us suppose that she marks her psychosis data as „sealed‟ and her termination data as „sealed and locked‟. All other data is „unsealed‟. This means that any data on the two sensitive areas in Alice‟s GP practice is available to any GP in that practice with whom Alice has a legitimate relationship i.e. any GP in that practice she decides to consult. However, the GP is made aware that Alice has marked the data as sensitive. The psychosis and termination data is also available to anyone in the corresponding departments of the hospital e.g. the psychiatric and obstetric departments respectively, with whom she has a legitimate relationship. Once again, they are made aware that it is marked sensitive.

The psychosis data is available to anyone outside the GP practice and psychiatric department, with whom Alice has a legitimate relationship, but access is strictly audited.

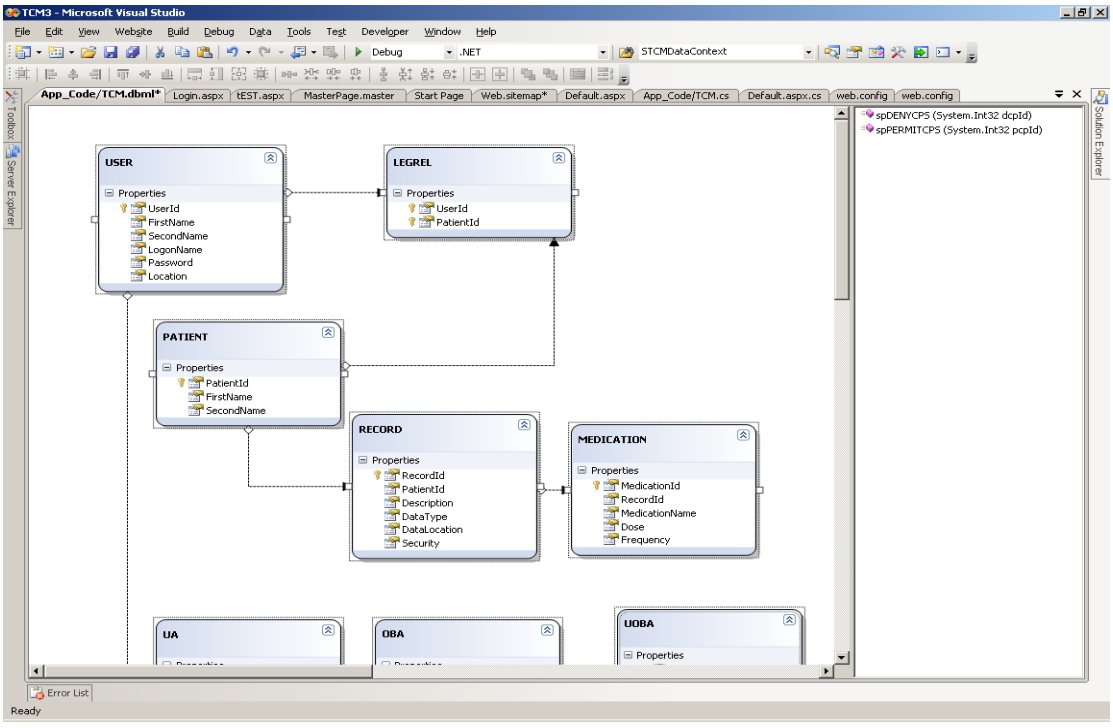
The termination data is not available, and indeed no one is aware of its existence, outside the GP practice, and the obstetrics department.

6.10.2 Additional Requirements

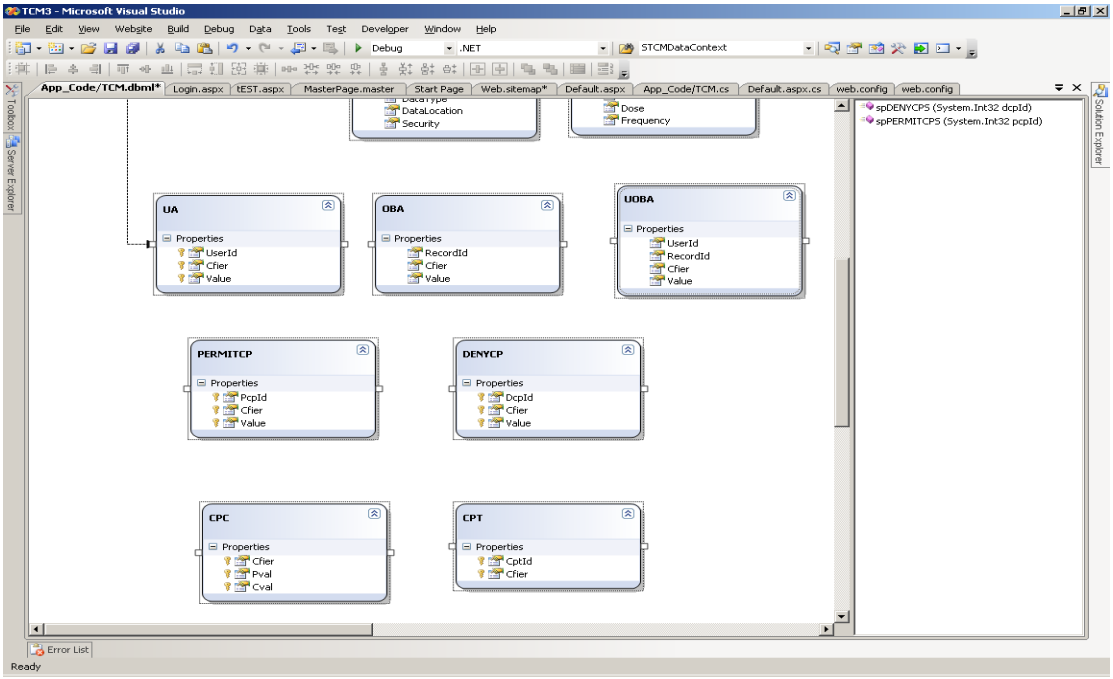
Let us suppose that Alice is referred to a psychiatric consultant at a different hospital e.g. from James Cook University Hospital to North Tees University Hospital, and Alice wishes them to see the termination data. Either she can reclassify the data as „sealed" instead of „sealed and locked", or the prohibition can be refined to allow the North Tees psychiatric consultant access. This second option would not be available with current models, but is available with the TCM. Reclassifying the data as „sealed" opens up the access to a wide range of users. TCM refinement makes an adjustment allowing access to a specific user.

6.10.3 LinQ to SQL dbml

The two screen shots below show how LinQ to SQL enables tables to be treated as objects with properties. The screen shots together show the LinQ to SQL designer.:



Screen 11: Sql Objects Top



Screen 12: Sql Objects Bottom

The CPT object in the above screen is only needed if implementing TCM2.

6.10.4 Stored Procedures

The versatility of LinQ means that only two stored procedures are required to implement TCM2 or TCM3. The two stored procedures give all the inherited values of a Confidentiality Permission. These stored procedures become methods when dragged onto the LinQ to SQL designer.

```
CREATE PROCEDURE [dbo].[spPERMITCPS]
@PcpId int

AS
BEGIN
WITH descs (Cfier,Value)
AS
(SELECT Cfier, Value from PERMITCPs
WHERE PcpId = @PcpId
UNION ALL
SELECT  c.Cfier, c.Cval from CPCS c
INNER JOIN descs on
            descs.Cfier = c.Cfier
AND         descs.Value = c.Pval
)
SELECT * FROM descs
END
```

```
CREATE PROCEDURE [dbo].[spDENYCPS]
@DcpId int

AS
BEGIN
WITH descs (Cfier,Value)
```

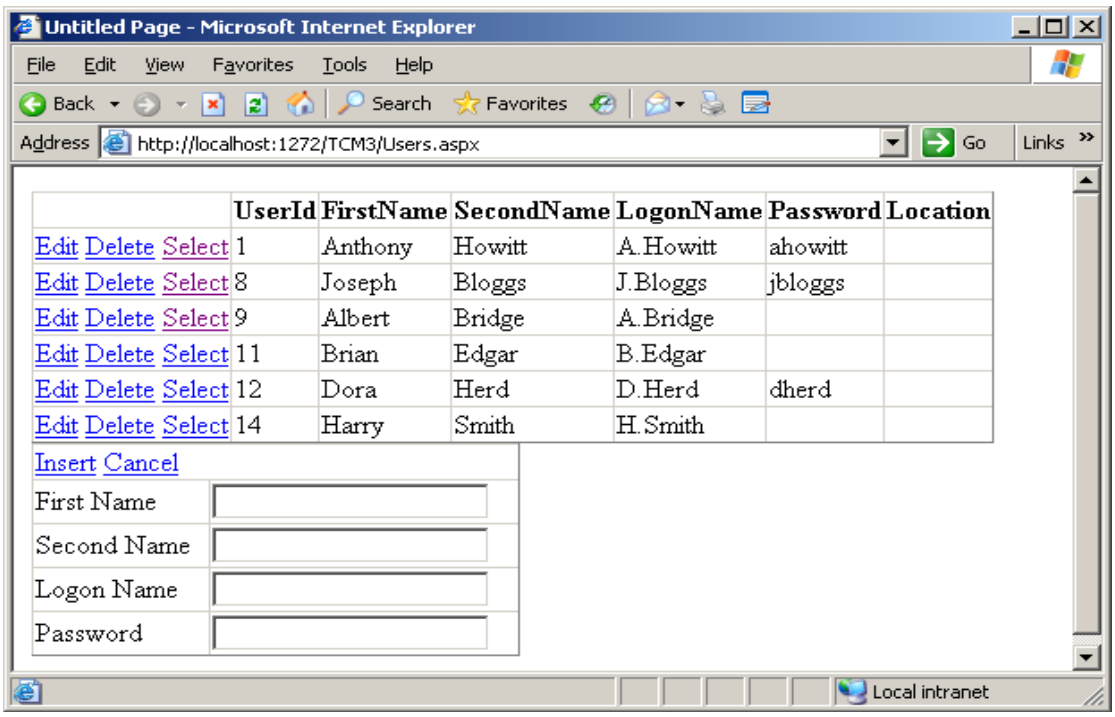
```
AS
(SELECT Cfier, Value from DENYCPs
WHERE DcpId = @DcpId
UNION ALL
SELECT  c.Cfier, c.Cval from CPCS c
INNER JOIN descs on
            descs.Cfier = c.Cfier
AND         descs.Value = c.Pval
)
SELECT * FROM descs
END
```

6.10.5 Implementation

The web pages below are to demonstrate the use of the TCM class to apply all the different TCM methods. The web pages demonstrate functionality. The data source for each GridView or other control is the TCM class. More sophisticated design layouts would be used in an application, whilst still using the same functionality.

6.10.6 Users

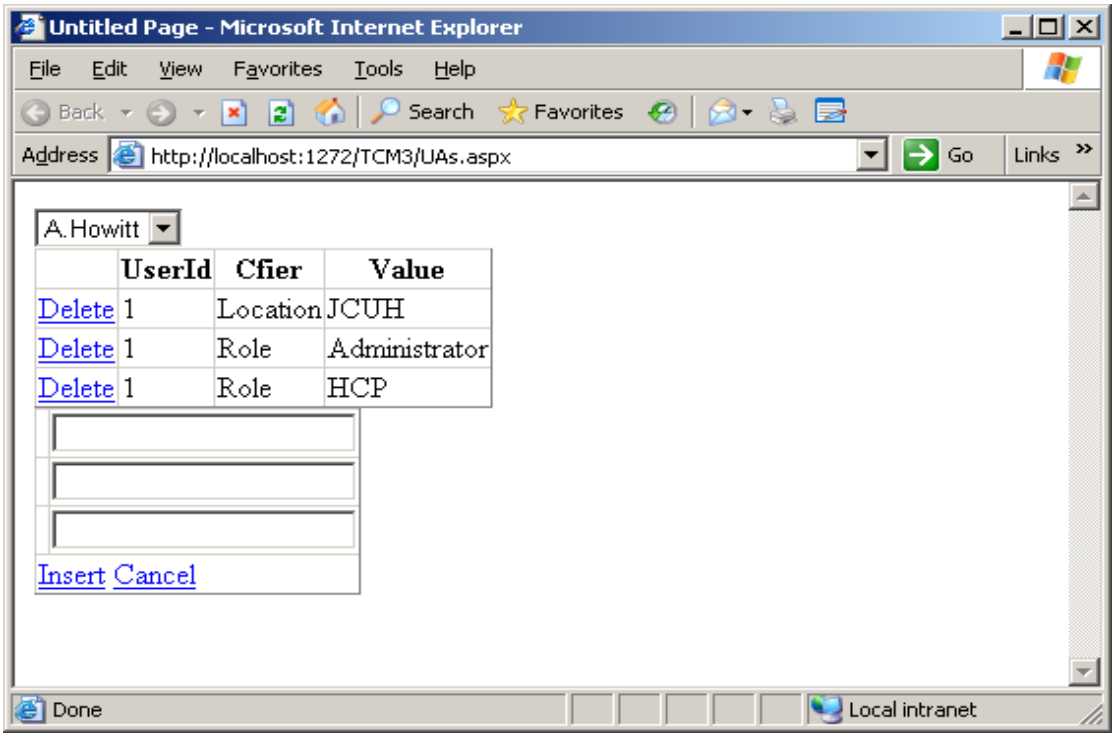
Users are stored using the TCM membership class which is a custom membership class overriding the asp.net membership class. All the features of the asp.net membership class are available, including the use of user profiles, and the ability to add additional user properties. Although an integer UserId and a plain text password is shown here, in an application use would be made of unique identifiers and encryption, both of which are available through the asp.net membership class.



Screen 13: Users

In the screen above, users can be added. A logon name and password can be allocated.

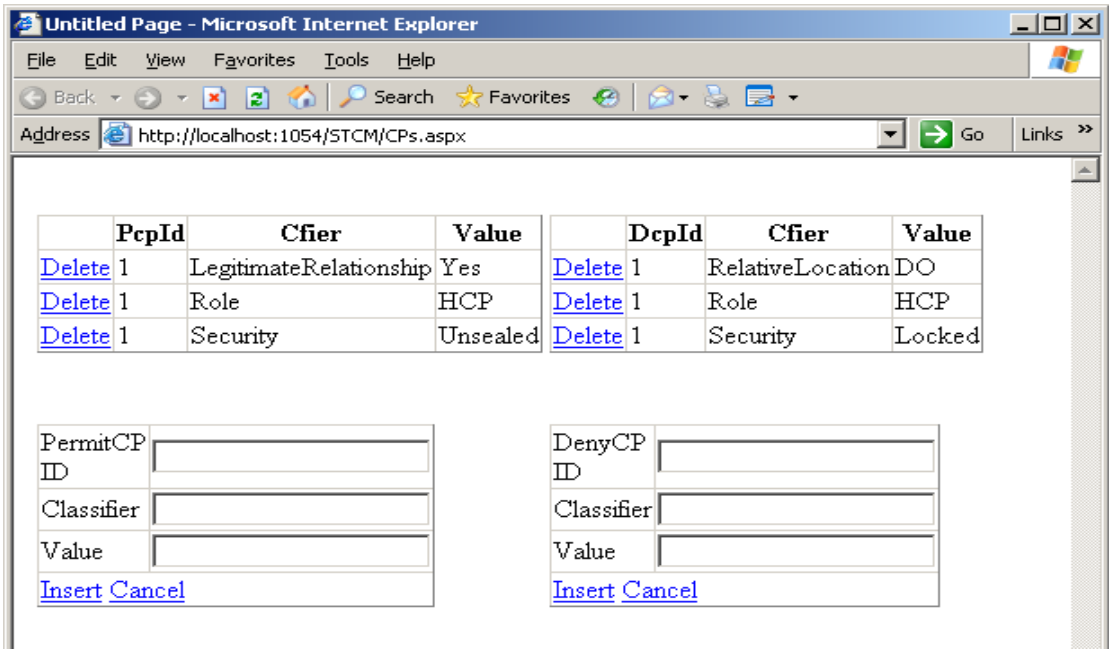
6.10.7 User Assignment



Screen 14: User Assignment

In the screen above, users can be assigned to different classifier values.

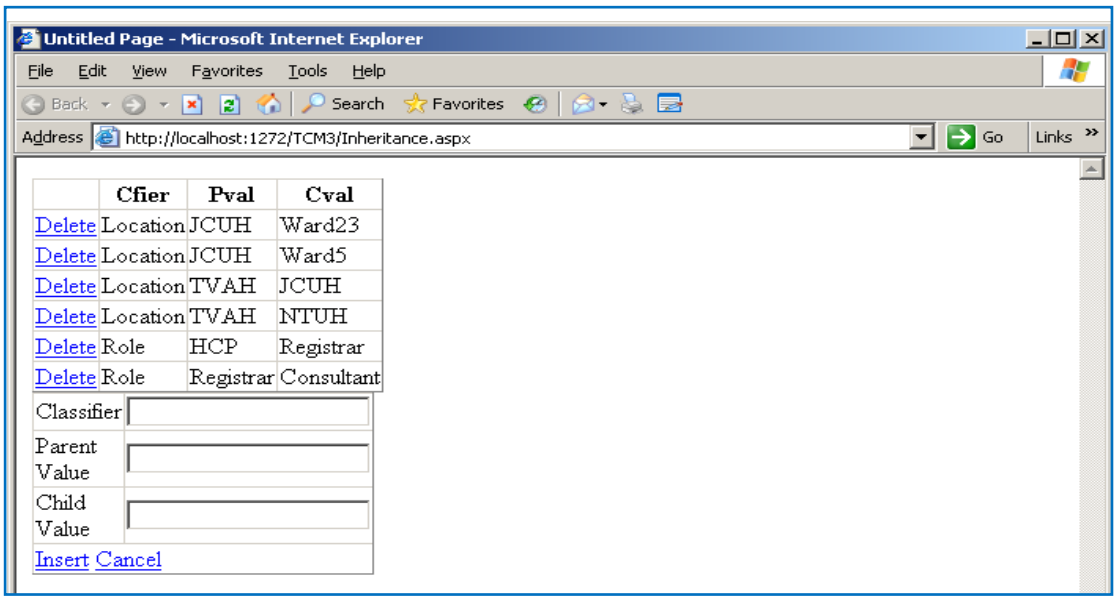
6.10.8 Confidentiality Permissions



Screen 15: Confidentiality Permissions

CPs are created using multiple classifier values in the above screen.

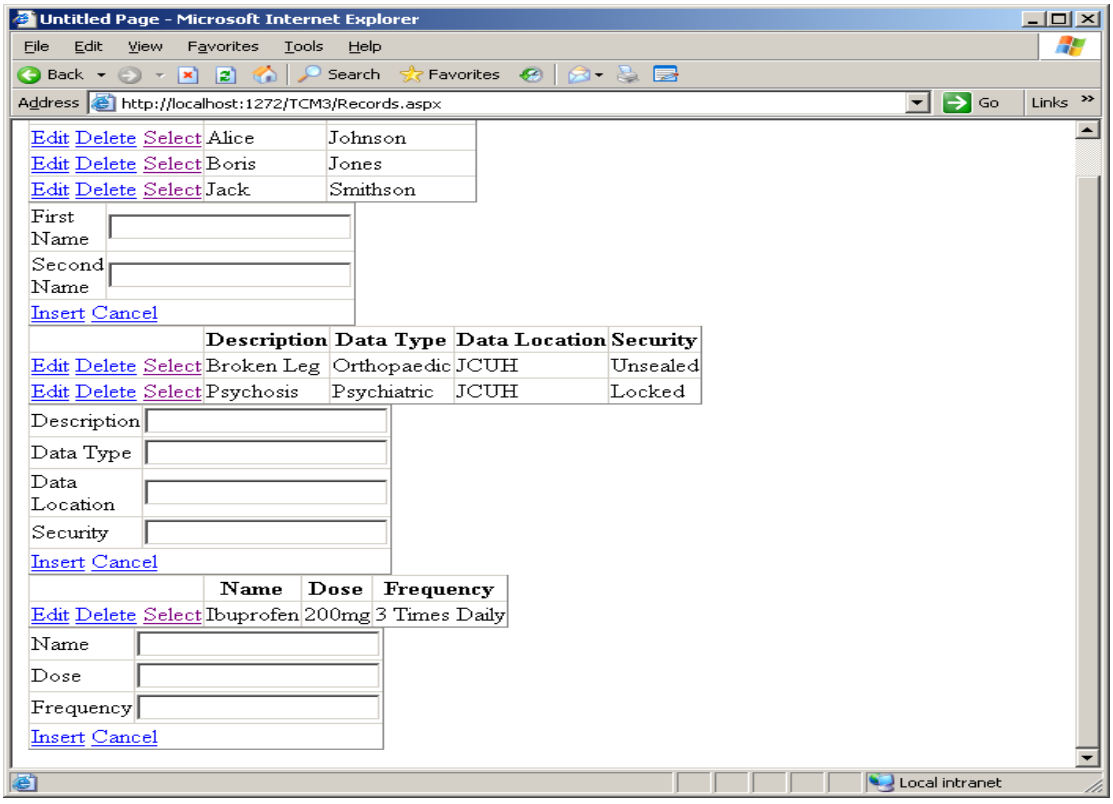
6.10.9 Inheritance



Screen 16: Inheritance

The above screen adds classifier to parent to child values.

6.10.10 Records

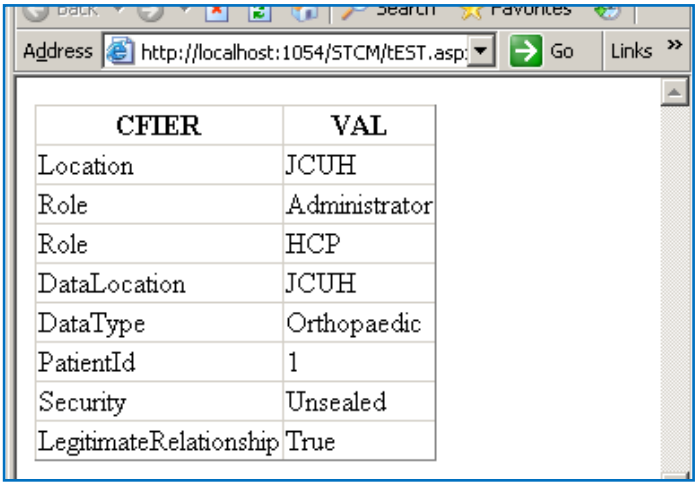


Screen 17: Patient Records

The screen above shows three tiers of patient records: Patient Details, Referral Description, and Drugs Prescribed.

6.10.11 Testing

The following set of Assigned Classifier Values belong to UserId = 1, and RecordId = 1. They are used to test whether the Confidentiality Permissions are working correctly.



CFIER	VAL
Location	JCUH
Role	Administrator
Role	HCP
DataLocation	JCUH
DataType	Orthopaedic
PatientId	1
Security	Unsealed
LegitimateRelationship	True

Screen 18: Active Classifier Values

These are used together with the inheritance relationship given by the table below.

Classifier	Parent	Child
Location	JCUH	Ward5
Location	JCUH	Ward23
Location	TVAH	JCUH
Location	TVAH	NTUH
Role	HCP	Registrar
Role	Registrar	Consultant

Table 2: Model Inheritance

The following table shows the tests and the results, which are as expected.

PermitCp	DenyCp	Permit Access	Deny Access	TCM PermitAccess
<Role,HCP> <Location,JCUH> <Security,Unsealed>	<Role,HCP> <Location,JCUH> <Security,Sealed>	True	False	True
<Role,HCP> <Location,JCUH> <Security,Unsealed>	<Role,HCP> <Location,JCUH> <Security,Unsealed>	True	True	False
<Role,HCP> <Location,JCUH> <Security,Unsealed>	<Role,Registrar> <Location,JCUH> <Security,Unsealed>	True	False	True
<Role,Registrar>	<Role,HCP>	False	True	False

<Location,JCUH> <Security,Unsealed>	<Location,JCUH> <Security,Unsealed>			
<Role,HCP> <Location,JCUH> <Security,Unsealed>	<Role,HCP> <Location,TVAH> <Security,Unsealed>	True	False	True
<Role,HCP> <Location,TVAH> <Security,Unsealed>	<Role,HCP> <Location,JCUH> <Security,Unsealed>	False	True	False
<Role,HCP> <Location,JCUH> <Security,Unsealed>	<Role,HCP> <Location,JCUH> <Security,Unsealed> <LegitimateRelationship,True>	False	True	False
<Role,HCP> <Location,JCUH> <Security,Unsealed> <LegitimateRelationship,True>	<Role,HCP> <Location,JCUH> <Security,Unsealed>	True	False	True
<Role,HCP> <Location,JCUH> <Security,Unsealed> <LegitimateRelationship,False>	<Role,HCP> <Location,JCUH> <Security,Unsealed>	False	True	False
<Role,HCP> <Location,JCUH> <Security,Unsealed> <LegitimateRelationship,False>	<Role,HCP> <Location,JCUH> <Security,Unsealed> <LegitimateRelationship,False>	False	False	False

Table 3: Test Results

6.11 Summary

Classes have been developed which enable the TCM to be applied in any authorisation scenario. The procedure to apply the classes is given in the appendix. For completeness, classes are given for both TCM2 and TCM3. A website has been developed which demonstrates the use of these classes, and some testing has been included.

7 Conclusions

7.1 Advances made to Access Control

Authorisation is a very important part of Identity and Access Management, yet has arguably not received the research and development attention that Authentication and Federated Identity Management have. It appears from the research literature that much of the development of authorisation has been influenced by Role Based Access Control, with many elaborations of this basic concept. The TCM, as originally proposed, and further developed in this thesis, is an attempt to investigate and develop authorisation from first principles.

Accordingly, the first contribution of the thesis is a thorough examination of RBAC. This was accomplished by developing a formal model (B Model) of the RBAC ANSI Standard, which itself was the culmination of many years of research. In the process of doing this, improvements were identified, and incorporated into the B Model. These improvements were possible because B enables the specification to be verified. There were many errors in the mathematics of the RBAC ANSI Standard that were identified and eliminated to provide a more robust model and one suitable for further development.

Following this, a B model of authorisation based on the original TCM was developed. During the course of this development, it became apparent that a fundamental improvement to the original TCM (referred to as TCM1 in the text) was desirable, namely replacing the concept of collection with classifier value hierarchy, with single inheritance. A formal framework with more general concepts than RBAC (e.g. „classifier" performing a more general function than „role") was developed.

The worth of these improvements was demonstrated in that they were used in ongoing work to develop a USA-based healthcare application of the TCM (by Dr Longstaff); the original TCM would have been unable to handle this demanding application. The impact of the original TCM, and particularly of the enhancements presented in this thesis, is that on the basis of seen publications, and other communications, Dr Longstaff was invited to sit on an Expert Panel on healthcare software development in the USA in January 2008 (ASPE 2008).

In addition to the above, the thesis reports work on developing a further, simplified version of the TCM, and implementing and applying it using the most modern developments in programming software. TCM3 uses Spec#, Spec Explorer and LinQ to develop classes

that can be applied to any authorisation scenario. For completeness classes for TCM2 have also been developed.

Therefore, the thesis has made a significant contribution to the theory and practice of access control, which has been recognised and applied in commercial settings.

7.2 Further Work

It is intended that the modelling work on the RBAC standard, together with the detailing of faults detected in the RBAC standard, be communicated to the ANSI INCITS technical committee. The basis of the work was presented at a B conference in 2008 (Dunne, et al., 2008).

Other further work would be the application of the TCM to various „real world" situations, and to continue to demonstrate that the TCM as described in this thesis has both the flexibility and the simplicity to cope with the most demanding authorisation scenarios.

The application of both TCM2 and TCM3 would help determine which of these models is most suitable for use in general authorisation scenarios. It is my belief that TCM3 can be shown to be powerful enough for all scenarios.

References

Abrial J.R. The B-Book: Assigning Programs to Meanings [Book]. - [s.l.] : Cambridge University Press, 1996.

AHIMA
http://library.ahima.org/xpedio/groups/public/documents/ahima/bok1_027539.hcsp?dDocName=bok1_027539 [Online].

Bacon J, Moody K and Yao W A Model of OASIS Role-Based Access Control and Its Support for Active Security [Journal] // ACM Transactions on Information and System Security, 5, 4.. - 2001. - 4 : Vol. 5.

Barnett Mike [et al.] The Spec# Programming System: An Overview [Journal]. - [s.l.] : Springer, 2004. - CASSIS 2004, LNCS : Vol. 3362.

Barnett Mike [et al.] Verification of Object-Oriented Programs with Invariants [Report]. - 2004. - JOT3(6).

B-Core B-Toolkit [Online] // www.b-core.com.

Campbell Colin [et al.] Model-Based Testing of Object-Oriented Reactive Systems [Report]. - [s.l.] : MSR-TR-2005-59, 2005.

Chakraborty S and Ray I TrustBAC - Integrating Trust Relationships into the RBAC Model for Access Control in Open Systems [Conference]. - [s.l.] : Proceedings of Eleventh ACM Symposium on Access Control Models and Technologies, 2006.

Department of Defence Trusted Computer System Evaluation Criteria. - [s.l.] : TCSEC, 1985. - DoD 5200.28 - STD.

Department of Health
http://www.dh.gov.uk/en/Publicationsandstatistics/Publications/PublicationsPolicyAndGuidance/DH_4106506 [Online] // Creating a patient-led NHS: Delivering the NHS Improvement Plan. - 17 March 2005.

Dunne Steve and Howitt Anthony Modelling Role-based Access Control in B [Conference] // Les Journées Scientifiques de l'Université de Nantes. - Nantes : [s.n.], 2008.

Ferraiola D F [et al.] Proposed NIST Standard for Role-Based Access Control [Journal]. - [s.l.] : ACM Transactions on Information and System Security, 2001. - 3 : Vol. 4.

Ge M and Osborn S A Design for Parameterized Roles [Article] // Data and Applications Security. - [s.l.] : Status and Prospect, 2004. - XVIII.

Goh C and Baldwin A Towards a more Complete Model of Role [Conference]. - [s.l.] : Proceedings of Third ACM Workshop on Role-Based Access Control, 1998.

Hall Anthony Realising the Benefits of Formal Methods [Online] // http://www.cs.man.ac.uk/~banach/CSIComm-May07-FM-Papers/080Hall.pdf. - 2006.

Healthspace [Online]. - www.healthspace.nhs.uk.

HealthVault www.healthvault.com [Online].

INCITS Role-Based Access Control. - [s.l.] : American National Standard for Information Technology, 2004. - ANSI INCITS 359-2004.

Indivo www.indivohealth.org [Online].

Infoway Electronic Health Record Infostructure (EHRi) Privacy and Security Conceptual Architecture [Journal]. - [s.l.] : Canada Health Infoway Inc., 2005.

Longstaff J J [et al.] Eliciting and recording eHR/ePR Patient Consent in the context of the Tees Confidentiality Model. [Conference] // Proceedings of HC2002 Conference. - Harrogate : [s.n.], 2002. - ISBN 0 9535427 6 9.

Longstaff J J, Lockyer M A and Howitt A Functionality and implementation issues for complex authorisation models [Journal]. - [s.l.] : IEE Proceedings - Software, February 2006. - 1 : Vol. 153. - ISSN 1462-5970.

Longstaff J J, Lockyer M A and Nicholas J An Authorisation Model for Complex Web Applications [Conference]. - Proceedings of Eighth ACM Symposium on Access Control : [s.n.], 2003.
Longstaff Jim, Lockyer Mike and Howitt Tony Functionality and implementation issues for complex authorisation models [Journal].
Mazzoleni P [et al.] XACML Policy Integration Alogorithms [Conference]. - [s.l.] : Proceedings of Eleventh ACM Symposium on Access Control Models and Technologies, 2006.
National Institute of Standards and Technology [Online] // http://www.nist.gov/.
Neumann G and Strembeck M A Scenario-driven Role Engineering Process for Functional RBAC Roles [Conference]. - [s.l.] : Proceedings of Seventh ACM Symposium on Access Control Models and Technologies, 2002.
NHS Care Records www.nhscarerecords.nhs.uk [Online].
NHS http://www.krha.nhs.uk/ER/EHR/Index.htm [Online] // Electronic Health Records and ERDIP. - Local Implementation Strategy (LIS), 2003.
NHS Portal
https://www.portal.nss.cfh.nhs.uk/sites/connectpatient/trainthetrainer/default.aspx [Online] // NHS Portal.
Office of the Assistant Secretary for Planning and Evaluation Expert Panel on Security Technologies [Conference] // Services, United States Department of Health and Human. - San Antonio : HL7 International privacy Architecture Meeting, 2008.
Oh S, Sandhu R and Zhang X An Effective Role Administration Model Using Organization Structure [Journal] // ACM Transactions on Information and System Security. - 2006. - 2 : Vol. 9.
Organization for the Advancement of Structured Information Standards Organization for the Advancement of Structured Information Standards [Online] // http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security. - 2006. - SAML standard.
Osborn S and Guo Y Modeling users in role-based access control [Conference] // Proceedings of Fifth ACM Workshop on Role-Based Access Control. - 2000.
Project Liberty Liberty Alliance Project Papers and Standards [Online] // www.projectliberty.org. - 2006.
Rattz Joseph C Pro LinQ: Language Integrated Query in C# 2008 [Conference]. - 2007.
Schneider S. The B-Method: An Introduction [Book]. - [s.l.] : Palgrave, 2001.
Science Applications International Corporation, SAIC
http://www.va.gov/rbac/docs/20050713_HL7_RBAC_Role_Engineering_Process_v1_0.doc [Online] // Role Based Access Control, Role Engineering Process v1.0.. - 2005.
Simons William W, Mandl Kenneth D and Kohane Isaac S The PING Personally Controlled Electronic Medical Record [Online]. - http://www.pubmedcentral.nih.gov/picrender.fcgi?tool=pmcentrez&blobtype=pdf&artid=543826.
Stermbeck M and Neuman G An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments [Journal]. - [s.l.] : ACM Transactions on Information and System Security, 2004. - 3 : Vol. 7.
The Independent European Association for eBusiness Users Guide to Role Based Access Control [Online] // www.eema.org. - 2003.
Thomas R K Term-Based Access Control (TMAC) [Conference] // Second ACM Workshop on Role- Based Access Control. - 1997.
Wilson D E and Clark D D A Comparison of Commercial and Military Security Policies [Report]. - [s.l.] : IEEE Symposium of Security and Privacy, 1987.

Zhang X [et al.] Formal Model and Policy Specification of Usage Control [Journal]. - [s.l.] : ACM Transactions on Information and System Security, 2005. - 4 : Vol. 8.

Appendices

A. Notation

P ∧Q	Conjunction: `` P and Q''.
P ∨Q	Disjunction: `` P or Q''.
P ⇒ Q	Implication: ``P implies Q'' or ``if P then Q''.
¬P	Negation: ``Not P''.
∀z.(Q => P)	Universal quantification: ``For all z where Q, P''.
∃z.P	Existential quantification: ``For some z, P holds''.
E ↦ F	Ordered pair (maplet).
E ∈ S	Set membership: the predicate ``E belongs to S''.
E ∉ S	Set non-membership: the predicate ``E does not belong to S''.
S ⊆ T	Set inclusion: the predicate ``S is included in T''.
S ⊈ T	Set non-inclusion: the negation of the predicate S ⊆ T.
S ∪ T	Set union: the set of elements which are elements of S or T.
S ∩ T	Set intersection: the set of elements which are elements of S and T.
φ	Empty set: the set with no elements.
F(S)	Finite subsets: Set of all finite subsets of S.
S ↔ T	Relation: Set of relations from S to T.
dom(r)	Domain of r: The set {x | x: S & ∃y.(x,y: r)}.
ran(r)	Range of r: The set {y | y: T & ∃x.(x,y: r)}.
p;q	Relational composition: Composition of relations p and q.
s ◁ r	Restriction of r by s. Also known as domain restriction.
r ▷ t	Co-restriction of r by t. Also known as range restriction.
s ⩤ r	Anti-restriction of r by s. Also known as domain subtraction.
r ⩥ t	Anti-co-restriction of r by t. Also known as range subtraction.
r^{-1}	Inverse of r.
r[s]	Image of set s under relation r.
S ⇸T	Set of partial functions from S to T
S → T	Set of total functions from S to T.
iseq(S)	The set of injective sequences of elements from S.
x ≔E	Simple substitution.
x ≔E || y ≔F	Simultaneous substitution.
$\mathbb{N}_1$	Non-zero natural numbers.
⋃	Generalised union.

A full listing is available at www.b-core.com

B. Acronyms and Abbreviations

ACL	Access Control List: a list of permissions attached to an object. The list specifies who or what is allowed to access the object and what operations are allowed to be performed on the object
AHIMA	American Health Information Management Association: an association of health information management professionals
ANSI	American National Standards Institute: a private organisation that oversees the development of voluntary consensus standards for products, services, processes, systems, and personnel in the United States.
CP	Confidentiality Permission: the fundamental mechanism for granting or denying permission in the TCM.
CPT	Confidentiality Permission Type: the type to which a CP belongs. Defined as the set of classifiers contained in a CP.
DAC	Discretionary Access Control: a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject
DSD	Dynamic Separation of Duties: the enforcement of rules regarding those duties (roles) that a user can activate at login
EEMA	European Electronic Messaging Association: an independent, trade association for e-Business, working to further e-Business technology and legislation with its European members, governmental bodies, standards organisations and e-Business initiatives.
EHR	Electronic Health Records: an individual patient's medical record in digital format
ERDIP	Electronic Records Development and Information Programme
INCITS	International Committee for IT Standards: an ANSI-accredited forum of IT developers

MAC	Mandatory Access Control: A system of access control that assigns security labels or classifications to system resources and allows access only to entities (people, processes, devices) with distinct levels of authorisation or clearance
NHSIA	National Health Service Information Authority: part of the UK National Health Service (NHS), established by an Act of Parliament in 1999, superseded in 2004.
NIST	National Institute of Standards and Technology: a non-regulatory agency of the United States Department of Commerce
OASIS	Organisation for the Advancement of Structured Information Standards: a global consortium that drives the development, convergence and adoption of e-business and web service standards
PHR	Personal Health Record: The personal health record (PHR) is an electronic, universally available, lifelong resource of health information needed by individuals to make health decisions. Individuals own and manage the information in the PHR, which comes from healthcare providers and the individual. The PHR is maintained in a secure and private environment, with the individual determining rights of access. The PHR is separate from and does not replace the legal record of any provider
RBAC	Role Based Access Control: an approach to granting system access to authorised users based on their roles
SAML	Security Assertion Markup Language: an XML-based standard for exchanging authentication and authorisation data between security domains
SSD	Static Separation of Duties: the enforcement of rules regarding those duties (roles) to which a user can activate be assigned
TCM	Tees Confidentiality Model: an authorisation model which is suitable for complex web applications
XACML	eXtensible Access Control Markup Language: a declarative access control policy language implemented in XML

XML	Extensible Markup Language: a general-purpose *specification* for creating custom markup languages

C. Relational Operators in Spec#

Domain

The domain of a set of classifier values

```
Set<CFIER> dom(Set<<CFIER,VALUE>> S)
{
  return Set{<c,v>in S; c};
}
```

Range

The range of a set of classifier values

```
Set<CFIER> ran(Set<<CFIER,VALUE>> S)
{
  return Set{<c,v> in S; v};
}
```

Domain Restriction

The set of classifier values in S restricted to the given set of classifiers C

```
Set<<CFIER,VALUE>> domRestriction(Set<CFIER> C, Set<<CFIER,VALUE>>
S) {
  return Set{c1 in C, <c2,v> in S, c1==c2; <c1,v>};
}
```

Domain Anti-Restriction

The set of classifier values in S restricted to classifiers not in the given set of classifiers C

```
Set<<CFIER,VALUE>> domAntiRestriction(Set<CFIER> C,
Set<<CFIER,VALUE>> S) {
  return Set{c1 in dom(S)-C, <c2,v> in S, c1==c2; <c1,v>};
}
```

Range Restriction

The set of classifier values in S restricted to the given set of values V

```
Set<<CFIER,VALUE>> ranRestriction(Set<VALUE> V, Set<<CFIER,VALUE>>
S) {
  return Set{<c,v1> in S, v2 in V, v1==v2; <c,v1>};
}
```

Range Anti-Restriction

The set of classifier values **in** S restricted to values not in the given set of values V

```
Set<<CFIER,VALUE>> ranAntiRestriction(Set<CFIER> V,
Set<<CFIER,VALUE>> S) {
  return Set{<c,v1> in S, v2 in ran(S)-V, v1==v2; <c,v1>};
}
```

Relational Image

The set of values in S corresponding to the given set of classifiers C

```
Set<VALUE> relImage(Set<CFIER> C, Set<<CFIER,VALUE>> S) {
  return Set{c1 in C, <c2,v> in S, c1==c2; v};
}
```

Relational Inverse

The inverse of relation R

```
Set<<VALUE,CFIER>> relInverse(Set<<CFIER,VALUE>> R) {
  return Set{<c,v> in R; <v,c>};
}
```

Relational Composition

The composition of two inheritance relationships R0 and R1 with the relationships partitioned in classifiers

```
Set<<CFIER,VALUE,VALUE>> relComposition(Set<<CFIER,VALUE,VALUE>> R0,
Set<<CFIER,VALUE,VALUE>> R1) {
  return Set{<c0,vp0,vc0> in R0, <c1,vp1,vc1> in R1,c0==c1,vc0==vp1;
<c0,vp0,vc1>};
}
```

Relational Override

Overriding one inheritance relationship R0 by another one R1, i.e. R0 applies outside the domain of R1; else R1 applies.

```
Set<<CFIER,VALUE,VALUE>> relOverride(Set<<CFIER,VALUE,VALUE>> R0,
Set<<CFIER,VALUE,VALUE>> R1) {
  return
  Set{<c0,vp0,vc0> in R0, <c1,vp1,vc1> in R0-
R1,c0==c1,vp0==vp1,vc0==vc1; <c0,vp0,vc0>} +
  Set{<c0,vp0,vc0> in R0, <c1,vp1,vc1> in
R1,c0==c1,vp0==vp1,vc0==vc1; <c1,vp1,vc1>};
}
```

Identity

Identity relation on R.

```
Set<<CFIER,VALUE,VALUE>> id(Set<<CFIER,VALUE,VALUE>> R)
```

```
{
  return Set{<c,vp,vc> in R;<c,vp,vp>};
}
```

Iterate

The nth iterate of relation R.

```
Set<<CFIER,VALUE,VALUE>> iterate(Set<<CFIER,VALUE,VALUE>> R, int n)
{
  switch (n)
  {
  case 0: return id(R);
  case 1: return R;
  default: n-=1;
  return Set{<c1,vp1,vc1> in iterate(R,n), <c2,vp2,vc2>
  in R,vc1==vp2; <c1,vp1,vc2>};
  }
}
```

Closure

The reflexive transitive closure of relation R.

```
Set<<CFIER,VALUE>> closure(Set<<CFIER,VALUE>> S,
Set<<CFIER,VALUE,VALUE>> R)
{
  var Set<<CFIER,VALUE>> clos = Set{};
  var Set<<CFIER,VALUE>> add = S;
  int i=1;
  while(add - clos > Set{})
  {
    clos = clos + add;
    add = Set{<c1,v>in S, <c2,pv,cv> in
iterate(R,i),c1==c2,v==pv;<c1,cv>};
    i+=1;
  }
  return clos;
}
```

D. Implementing a Web Application Using the TCM

1. Create a new website using Visual Studio 2008.
2. Add the TCM class.
3. Add the TCM2 or TCM3 class depending on which version you are using.
4. Add the TCM database to the App_Data folder. (Alternatively, attach to another SQL Server instance and modify the connection string accordingly).
5. Add a new LinQ to SQL TCM data class: .dbml file.
6. From Server Explorer drag the tables, views and stored procedures onto the TCM.dbml designer.
7. Use the TCM methods in the development of your application.

Additionally there is a TCM membership class that can be used in the application. Asp.net‟s own membership class can be used or any other membership provider such as Active Directory. The TCM membership class can be further customised to suit individual requirements. To use TCM membership:

8. Add TCM membership class.
9. In the web.config file, change the authentication mode. <authentication mode="Forms"/>
10. Just below <authentication mode="Forms"/> add the following.

```
<membership defaultProvider = "TCMMembershipProvider"
     UserIsOnlineTimeWindow="15">
    <providers>
     <add name="TCMMembershipProvider"
       type="TCMMembershipProvider"
       enablePasswordRetrieval="false"
       enablePasswordReset="false"
       requiresQuestionAndAnswer="false"
       QuestionRequired ="false"
       applicationName="/"
       requiresUniqueEmail="false"
       passwordFormat="Clear"
       description="Stores and retrieves membership data from SQL Server"
       decryptionKey="68d288624f967bce6d93957b5341f931f73d25fef798ba75"
```

```
      validationKey="65a31e547b659a6e35fdc029de3acce43f8914cb1b2
                     4fff3e1aef13be438505b3f5becb5702d15bc7b98cd
                     6fd2b7702b46ff63fdc9ea8979f6508c82638b129a"
    />
  </providers>
</membership>
```

11. Customise as required in web.config and the membership class.